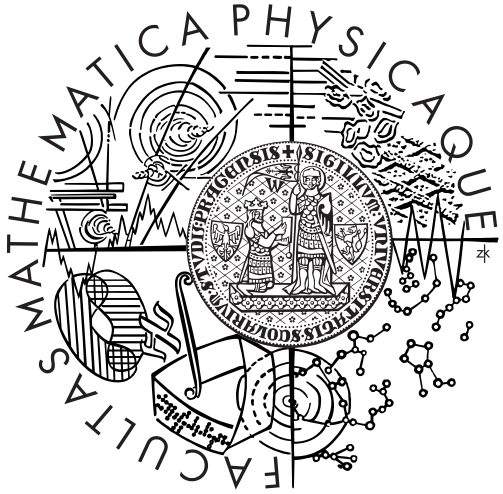
Univerzita Karlova v Prahe
Matematicko-fyzikálna fakulta

DIPLOMOVÁ PRÁCA



Patrik Gebrian

Unicode Compiler-Compiler

Katedra softwarového inžinierstva
Vedúci diplomovej práce: RNDr. Leo Galamboš, Ph.D.
Študijný program: Informatika

Rád by som na tomto mieste poďakoval svojmu vedúcemu RNDr. Leovi Galambošovi, Ph.D. za množstvo cenných rád a pripomienok a za poskytnutie HTML dát na testovanie.

Prehlasujem, že som svoju diplomovú prácu napísal samostatne a výhradne s použitím citovaných prameňov. Súhlasím s požičiavaním práce.

V Prahe dňa 7.augusta 2006. Patrik Gebrian

Obsah

Názov práce: Unicode Compiler-Compiler
Autor: Patrik Gebrian
Katedra: Katedra softwarového inžinierstva
Vedúci diplomovej práce: RNDr. Leo Galamboš, Ph.D.
e-mail vedúceho: Leo.Galambos@mff.cuni.cz

Abstrakt: Cieľom tejto práce je vytvoriť efektívny LALR(1) generátor parserov pre jazyk Java, ktorý by mal byť ekvivalentný k existujúcim produktom Flex a Bison pre jazyk C. V práci sú analyzované existujúce LL(k) a LALR(1) generátory parserov pre jazyk Java z pohľadu využitia objektových technológií, ako je vytváranie objektov a garbage collector. Na základe tejto analýzy je navrhnutý a implementovaný nový generátor parserov Achilex/Paris, založený na existujúcich produktoch JFlex a CUP. Implementácia používa efektívne techniky na prácu s objektmi, ako je napríklad object pooling. Dôležitou súčasťou práce je porovnanie implementovaného generátora parserov s produktmi JavaCC a JFlex/CUP, na jazykoch HTML a Java.

Kľúčové slová: generátor parserov, garbage collector, object pooling, HTML

Title: Unicode Compiler-Compiler
Author: Patrik Gebrian
Department: Department of Software Engineering
Supervisor: RNDr. Leo Galamboš, Ph.D.
Supervisor's e-mail address: Leo.Galambos@mff.cuni.cz

Abstract: The aim of this work is to create an effective LALR(1) parser generator for Java language, that should be equivalent to the existing tools for C language, Flex and Bison. Usage of the object technologies like the object creation and the garbage collector has been analyzed for existing LL(k) and LALR(1) parser generators for Java language. As the result of analysis, new parser generator Achilex/Paris based on existing tools JFlex and CUP has been designed and developed. One of the effective techniques that is used by Achilex/Paris is object pooling. Important part of this work is also comparison with existing tools JavaCC and JFlex/CUP on two languages, HTML and Java.

Keywords: parser generator, garbage collector, object pooling, HTML

1 Úvod

Pod pojmom *parser* rozumieme program, prípadne časť programu, ktorého úlohou je analýza štruktúry vstupných dát a ich transformácia do dátových štruktúr, vhodných na ďalšie spracovanie. Parsery sa v určitej podobe vyskytujú v podstate vo všetkých programoch. Či už sa jedná o jednoduché spracovanie príkazov z príkazovej riadky alebo spracovanie zdrojových kódov komplexného programovacieho jazyka. Proces vytvárania parsera je možné automatizovať použitím generátora parserov. *Generátor parserov* je program, ktorý na základe špecifikácie štruktúry dát, doplnenej o programátorom definované obslužné rutiny, takzvané *akcie,* vygeneruje zdrojový kód parsera. Automaticky vytvorená časť kódu realizuje analytickú funkciu parsera, akcie realizujú transformačnú funkciu parsera.

Pôvodné generátory parserov boli určené pre jazyk C. S rastom popularity objektovo orientovaného programovania, sa objavili generátory parserov pre jazyky ako C++ alebo Java. Využitie objektovo orientovaných prostriedkov, umožňuje na jednej strane vytvoriť flexibilný, ľahko udržiavateľný software, no na druhej strane, so sebou prináša nevyhnutnú réžiu, ktorá vedie k vyšším pamäťovým a časovým nárokom.

Cieľom tejto práce je analyzovať existujúce generátory parserov pre programovací jazyk Java, odhaliť použitie operácií, ktoré sú v Jave neefektívne a na základe týchto znalostí navrhnúť a implementovať nový generátor parserov pre Javu, ktorý bude ekvivalentný produktom Flex a Bison pre jazyk C.

Práca začína všeobecným popisom generátorov parserov. V tretej kapitole sú uvedené objektové technológie využívané v jazyku Java, so zameraním na garbage collector. Kapitola 4 prináša analýzu vybraných objektových technológií, formou porovnania na testovacích programoch. V nasledujúcich dvoch kapitolách je uvedený popis a analýza štruktúry vygenerovaného parsera pre dva populárne generátory parserov pre jazyk Java. Kapitola 5 sa zaoberá LL(k) generátorom JavaCC. V kapitole 6 je popísaná dvojica nástrojov na generovanie lexikálnych a LALR(1) syntaktických analyzátorov, JFlex a CUP. V kapitole 7 je popísaný návrh vlastného generátora parserov. V nasledujúcej kapitole je uvedená jeho implementácia. Kapitola 9 prináša porovnanie analyzovaných generátorov parserov s vlastnou implementáciou. Záverečné zhodnotenie je uvedené v kapitole 10.

Súčasťou práce je takisto priložené CD s vlastnou implementáciou generátora parserov. Popis CD je uvedený v prílohe.

1.1 Testy

Dôležitou súčasťou práce je aj porovnanie na testovacích programoch. Testy prebehli na počítači s konfiguráciou Intel Celeron M 1.5GHz, 512MB RAM. Na čítanie dát bol použitý 120GB HDD so 7200 otáčkami. Na zápis dát bol použitý 40GB HDD so 4200 otáčkami. Použitý bol operačný systém Microsoft Windows XP SP2 a Java platforma vo verzii 1.5.0_06. Počas behu testov bol počítač vypnutý z počítačovej siete a počet bežiacich procesov minimalizovaný. Na získanie telemetrických údajov využitia haldy, bol použitý profilovací nástroj JProfiler vo verzii 4.2.1.

2 Generátory parserov

2.1 Základy

Ako už bolo spomenuté v úvode, vstupom generátora parserov je špecifikácia štruktúry spracovávaných dát. Tá sa skladá z nasledujúcich častí:

- **Špecifikácia lexikálnej štruktúry dát** - Je realizovaná prostredníctvom regulárnych výrazov, kde každý regulárny výraz definuje množinu reťazcov, takzvaných *lexémov*, ktoré sa môžu vyskytnúť vo vstupných dátach. Ku každému regulárnemu výrazu môže programátor definovať akciu, ktorá sa vykoná, keď dôjde k rozpoznaniu lexému, patriaceho do množiny lexémov daného regulárneho výrazu. Rozpoznanie lexému sa často označuje ako *zhoda*. Kód vygenerovaný z tejto špecifikácie spolu s akciami sa nazýva *lexikálny analyzátor*. Okrem rozpoznávania lexémov a spúšťania akcií, je dôležitou úlohou lexikálneho analyzátora generovanie objektov reprezentujúcich lexémy, takzvaných *tokenov*, ktoré sú spracovávané syntaktickým analyzátorom. Tokeny predstavujú rozhranie medzi lexikálnym a syntaktickým analyzátorom. V závislosti na použitom generátore, je buď kód na generovanie tokenov vygenerovaný automaticky alebo je súčasťou akcií. Základným atribútom tokena je jeho identifikátor. Napríklad špecifikácia pre rozpoznávanie prirodzených čísel by mohla vyzerať takto:

  ```
  [0-9]+ {return new Token(NUMBER);}
  ```

  V zložených zátvorkách je umiestnený kód akcie, ktorá vygeneruje token s identifikátorom `NUMBER`.

- **Špecifikácia syntaktickej štruktúry dát** - Je realizovaná prostredníctvom gramatiky. Názvy terminálov v pravidlách gramatiky odkazujú na identifikátory tokenov, ktoré sú produktom lexikálneho analyzátora. Ku každému pravidlu gramatiky môže programátor definovať akciu, ktorá sa vykoná, keď dôjde k aplikácii pravidla na spracovávané dáta. Kód vygenerovaný z tejto špecifikácie spolu s akciami sa nazýva *syntaktický analyzátor*. Syntaktický analyzátor teda prijíma vstup v podobe tokenov, snaží sa na neho aplikovať pravidlá gramatiky a pri aplikácii pravidla volá odpovedajúcu akciu.

Oproti naprogramovanému parseru, má automaticky vygenerovaný nasledujúce výhody:

- **Úspora času** - V špecifikácii nie je riešená implementácia parsera. To urýchľuje prácu programátora a znižuje pravdepodobnosť výskytu chýb, ktoré sa ťažko odhaľujú, pretože ich výskyt závisí na povahe vstupných dát. Na odhaľovanie chýb samotnej špecifikácie, umožňujú generátory parserov zapnúť ladiaci režim, v ktorom sú evidované jednotlivé kroky parsera.
- **Zrozumiteľnosť** - Špecifikácia je zrozumiteľná každému, kto ovláda regulárne výrazy a gramatiky, ktoré predstavujú intuitívny prístup k popisu štruktúry. Oproti tomu, zorientovať sa v konštrukciách naprogramovaného parsera, môže byť náročné.
- **Znovupoužiteľnosť** - Napríklad pri spracovávaní zdrojových kódov nejakého

známeho programovacieho jazyka, bude pravdepodobne voľne dostupná jeho špecifikácia, ktorú stačí doplniť o akcie. Pri programovaní parsera, je nutné túto špecifikáciu naprogramovať.

- **Modifikovateľnosť** - Pri zmene štruktúry dát, stačí prepísať špecifikáciu. Naprogramovaný parser je nutné preprogramovať.
- **Eliminácia nejednoznačnosti** - V prípade, že špecifikácia obsahuje nejednoznačnú gramatiku, je táto skutočnosť programátorovi oznámená. Nejednoznačnosti v naprogramovanom parsery, musí programátor odhaliť sám.

2.2 Typy generátorov parserov

Generátory lexikálnych a syntaktických analyzátorov môžu byť implementované ako:

- **Samostatné programy** - výhodou rozdelenia do samostatných programov, je možnosť definovať v generátore lexikálnych analyzátorov rozhranie, ktorým bude lexikálny analyzátor komunikovať so syntaktickým analyzátorom a tak možnosť vygenerovať z jedného nástroja lexikálne analyzátory, schopné komunikovať s rôznymi syntaktickými analyzátormi.
- **Integrované v jednom programe** - pri integrácii do jediného programu je rozhranie pevne dané. Špecifikácia lexikálneho a syntaktického analyzátora je umiestnená v jednom súbore. Niektoré takto implementované generátory parserov, používajú pre lexikálnu a syntaktickú analýzu rovnaký algoritmus a umožňujú tak v lexikálnom analyzátore používať prostriedky syntaktickej analýzy.

2.2.1 Generátory lexikálnych analyzátorov

Vygenerovaný lexikálny analyzátor sa typicky skladá z nasledujúcich častí:

- **Ovládač vstupu -** poskytuje ostatným komponentom prístup k znakovým dátam vstupu.
- **Výkonná jednotka -** jej úlohou je rozpoznávať vo vstupných dátach lexémy, odpovedajúce zadaným regulárnym výrazom. Väčšinou býva implementovaná ako konečný automat. Na vytvorenie konečného automatu, sa používajú štandardné algoritmy z prostredia teórie jazykov. Kód lexikálneho analyzátora, implementovaného pomocou konečného automatu, nie je možné jednoducho modifikovať. Práve z tohto dôvodu existujú generátory, ktoré na implementáciu výkonnej jednotky, nepoužívajú konečný automat, ale generujú programové štruktúry, ktoré sú ľahko čitateľné a výsledný lexikálny analyzátor je tak možné jednoducho modifikovať a prispôsobovať.
- **Akcie -** kód, ktorý sa vykoná, keď dôjde k rozpoznaniu lexému. Programátor má v akcii prístup k rozpoznanému lexému. V akcii je typicky umiestnený kód na vygenerovanie tokena. Okrem základného atribútu tokena, identifikátora, je ďalším dôležitým atribútom tokena hodnota. Tá je typicky odvodená z lexému. Napríklad vygenerovanie tokena, reprezentujúceho prirodzené číslo, by mohla vyzerať takto:

  ```
  [0-9]+ {return new Token(NUMBER, toInt(yytext));}
  ```

  Premenná `yytext` reprezentuje lexém a funkcia `toInt` prevedie lexém na číselnú reprezentáciu.

2.2.2 Generátory syntaktických analyzátorov

Vstupom generátora syntaktických analyzátorov je špecifikácia, ktorá obsahuje pravidlá gramatiky zapísané v notácii BNF (*Backus-Naur Form*), prípadne EBNF (*Extended BNF*) a k nim priradené akcie. Vygenerovaný syntaktický analyzátor, pozostáva z nasledujúcich komponent:

- **Správa tokenov** - udržuje tokeny získané od lexikálneho analyzátora.
- **Výkonná jednotka** - realizuje deriváciu gramatiky podľa analyzovaných dát. V prípade, že existuje viacero výpočtových ciest, je na rozhodnutie o výbere výpočtovej cesty, použitá aktuálna sekvencia tokenov na vstupe, takzvaný *výhľad*. V prípade, že nie je možné rozhodnúť podľa výhľadu, je vstupná gramatika pre daný syntaktický analyzátor nejednoznačná. Podľa implementácie výkonnej jednotky, sa v súčasnosti rozdeľujú generátory syntaktických analyzátorov na LL(k) generátory a LALR(1) generátory.
- **Zásobník** - pomocná dátová štruktúra na uchovávanie symbolov gramatiky, prípadne stavu analyzátora.
- **Akcie -** generátor pracuje s dvoma druhmi symbolov: s terminálmi a neterminálmi. Ku každému symbolu je určený názov a typ. Terminály odpovedajú tokenom z lexikálnej analýzy. Názov terminálu odpovedá identifikátoru tokena a typ terminálu odpovedá typu hodnoty tokena. V akcii má programátor prístup k hodnotám symbolov na pravej strane pravidla a môže priradiť hodnotu neterminálu na ľavej strane pravidla.

LL(k) analyzátory

LL(k) analyzátory analyzujú vstup zľava doprava a produkujú najľavejšiu deriváciu. Symbol *k* označuje dĺžku výhľadu v počte tokenov.

Syntaktická analýza začína vložením začiatočného neterminálu na zásobník. V každej ďalšej fáze prebiehajú nasledujúce kroky. V prípade, že je na vrchole zásobníka terminál, porovná sa s aktuálnym tokenom na vstupe. V prípade zhody, je ako aktuálny označený nasledujúci token na vstupe a terminál sa zahodí. Inak je oznámená chyba. V prípade, že je na vrchole zásobníka neterminál, nahradí sa pravou stranou pravidla, ktoré má tento neterminál na ľavej strane. V prípade existencie viacerých pravidiel s týmto neterminálom na ľavej strane, sa na základe výhľadu musí určiť jedno pravidlo. V prípade, že nie je možné vybrať žiadne pravidlo, je oznámená chyba. Špeciálnym prípadom je LL(0) analyzátor, ktorý zakazuje použitie viacerých pravidiel s rovnakým neterminálom na ľavej strane. Inak by nebolo možné rozhodnúť, ktoré pravidlo použiť.

LALR(1) analyzátory

LALR(1) analyzátory patria do rodiny LR(k) analyzátorov. LR(k) analyzátory analyzujú vstup zľava doprava a produkujú najpravejšiu deriváciu odzadu. Symbol *k* označuje dĺžku výhľadu, v počte tokenov.

Syntaktický analyzátor používa dve akcie. Akciu *posunutia* a *redukcie*. Pri akcii posunutia, je aktuálny token na vstupe vložený na zásobník a vstup sa posunie na nasledujúci token. Pri akcii redukcie, sa sekvencia terminálov a neterminálov na vrchole zásobníka, nahradí neterminálom na ľavej strane pravidla, ktorého pravá strana odpovedá tejto sekvencii. Celá analýza je realizovaná vykonávaním týchto dvoch akcií. Na

rozhodovanie o použití redukcie prípadne posunutia, je použitý konečný automat. Problém nastáva, keď v jednom stave analyzátora je možné súčasne vykonať akciu posunutia a redukcie, prípadne akciu redukcie podľa viacerých pravidiel. V takejto situácii sa musí použiť na rozhodnutie výhľad. Neschopnosť rozhodnúť sa medzi akciou posunutia a redukcie, sa označuje ako *s/r konflikt*. Neschopnosť vybrať pravidlo, podľa ktorého sa akcia vykoná, sa označuje ako *r/r konflikt*. V LR(0) analyzátoroch, môže byť v jednom stave analyzátora prípustná iba akcia posunutia alebo redukcie podľa jediného pravidla.

Porovnanie analyzátorov – všeobecnosť

LL(k) gramatiky neumožňujú niektoré konštrukcie zápisu, ktoré sú pomerne bežné pre BNF notáciu. Jedná sa o nasledujúce konštrukcie:

- **Ľavá rekurzia -** napríklad pre dvojicu pravidiel

  ```
  A ::= Aa | a
  ```

  analyzátor nemôže predpokladať dĺžku reťazca zloženého zo symbolov `a` a teda nie je schopný rozhodnúť, kedy má použiť pravidlo `A ::= a`, ktoré ukončuje reťazec. Tento problém sa rieši využitím EBNF notácie, ktorá zavádza do gramatiky symboly `?`, `*`, `+` známe z prostredia regulárnych výrazov. Teda pôvodná dvojica pravidiel by sa v EBNF notácii zmenila na jediné pravidlo:

  ```
  A ::= (a)+
  ```

  Prepis gramatiky s ľavou rekurziou do EBNF notácie, je väčšinou jednoduchý a výsledná gramatika môže byť dokonca prehľadnejšia, čo však záleží aj na vkuse. Je možné odstránenie ľavej rekurzie so zachovaním prostriedkov BNF notácie. Takáto transformácia však väčšinou vedie k neprehľadnej gramatike.

- **Spoločný prefix** - uvažujme nasledujúcu množinu pravidiel.

  ```
  A ::= B | C
  B ::= (a)+ b
  C ::= (a)+ c
  ```

  Podobne ako v predchádzajúcom prípade, analyzátor nemôže predpokladať dĺžku reťazca zloženého zo symbolov `a`, a teda nemôže rozhodnúť, či má použiť pravidlo `B` alebo `C`. Jedným z možných riešení je prepis gramatiky s využitím *ľavej faktorizácie*. Po použití tejto techniky, by mala predchádzajúca množina pravidiel nasledujúci tvar:

  ```
  A ::= (a)+ (b | c)
  ```

  Takáto transformácia však nemusí byť vždy jednoduchá a výsledná gramatika stráca na prehľadnosti. LL(k) generátory riešia tento problém zavedením takzvaných syntaktických predikátov. *Syntaktický predikát* je podmienka, ktorá musí byť splnená pred tým, ako je pravidlo použité. Podmienka sa zadáva formou pravidla. Po použití syntaktického predikátu, by predchádzajúca množina pravidiel vyzerala takto:

```
A ::= L(B) B | C
B ::= (a)+ b
C ::= (a)+ c
```

Analyzátor sa najprv pokúsi použiť pravidlo ohraničené funkciou `L`. V prípade, že uspeje, použije prvé pravidlo, inak použije druhé pravidlo. V prípade, že je dĺžka prefixu pevne daná, je možné tento problém vyriešiť aj zväčšením výhľadu. Práve kvôli spoločnému prefixu, je veľkosť výhľadu pre LL(k) analyzátory veľmi dôležitá. Použitie syntaktických predikátov, prípadne zväčšovanie výhľadu, pôsobí negatívne na výkon analyzátora.

LL(k) generátory zavádzajú do gramatík novú konštrukciu v podobe sémantických predikátov. *Sémantický predikát* je ľubovoľná booleovská podmienka, ktorá musí byť splnená pred tým, ako dôjde k použitiu pravidla. Použitím sémantických predikátov, je možné spracovávať aj kontextové gramatiky.

Porovnanie analyzátorov – jednoduchosť

Kód analyzátora vygenerovaný z LL(k) generátora je prehľadný a umožňuje jednoduché vykonávanie modifikácií. Vygenerovaný analyzátor sa skladá zo sady metód, kde každá metóda reprezentuje neterminál gramatiky. Vloženie pravej strany pravidla na zásobník, je simulované zavolaním metódy, ktorá reprezentuje neterminál na ľavej strane pravidla. Samotný kód metódy rozhoduje, v prípade existencie viacerých pravidiel s rovnakým neterminálom na ľavej strane, o výbere konkrétneho pravidla, a stará sa o spracovanie pravej strany vybraného pravidla. Teda terminály porovnáva s tokenmi na vstupe a pre neterminály volá odpovedajúce metódy. Veľká časť analyzátora je tak implementovaná použitím konštrukcií programovacieho jazyka.

Oproti tomu LALR(1) analyzátory sú implementované pomocou explicitného zásobníka a automatu, čo je v podstate tabuľka čísel. Takáto implementácia neumožňuje modifikácie a je náročnejšia na pochopenie.

Porovnanie analyzátorov – akcie

V LALR(1) analyzátoroch je akcia priradená k pravidlu vykonaná, keď dôjde k použitiu operácie redukcie podľa tohto pravidla. V akcii je programátorovi umožnený prístup k hodnotám terminálov a neterminálov pravidla umiestnených na zásobníku. V akcii môže programátor priradiť hodnotu neterminálu na ľavej strane pravidla. Po vykonaní akcie, sa pravá strana pravidla na vrchole zásobníka, nahradí neterminálom s priradenou hodnotu na ľavej strane pravidla.

V LL(k) analyzátoroch je kód akcie súčasťou tela metódy, ktorá reprezentuje neterminál na ľavej strane pravidla. Hodnoty terminálov a neterminálov sú predané pomocou lokálnych premenných metódy. Priradenie hodnoty neterminálu na ľavej strane pravidla, je realizované priradením návratovej hodnoty metóde. Okrem štandardného predávania informácií zdola nahor je možné pomocou parametrov metód, predávať informácie aj zhora nadol.

LALR(1) analyzátory predpokladajú umiestnenie akcií za pravou stranou pravidla. Pre akcie umiestnené v pravidle, generátor analyzátorov vykoná jednoduchú transformáciu gramatiky, pri ktorej sú akcie presunuté do automaticky vygenerovaných pravidiel

s prázdnou pravou stranou. Uvažujme dvojicu pravidiel:

```
A ::= {print(“nasleduje b”);} ab
A ::= ac
```

Po transformácii by dvojica pravidiel vyzerala takto:

```
A ::= Nab
A ::= ac
N ::= {print(“nasleduje b”);}
```

Výsledkom pokusu o vygenerovanie analyzátora z takejto gramatiky, by však bolo chybové hlásenie o s/r konflikte. Takáto gramatika je pre LALR(1) analyzátory nejednoznačná. Keď analyzátor vidí na vstupe symbol `a`, nevie rozhodnúť, či má použiť redukciu podľa tretieho pravidla alebo posunutie, ktoré umožňuje druhé pravidlo. Umiestnenie akcie do pravidla, teda môže zaviesť do gramatiky nejednoznačnosť.

Oproti tomu, v prípade LL(k) analyzátorov je nutné o použití pravidlá rozhodnúť pred tým, ako je známa jeho pravá strana. Vyššie uvedené umiestnenie akcie teda nevadí, ale na to aby bolo možné rozhodnúť o tom, ktoré pravidlo bude použité, je nutné použiť výhľad o dĺžke 2.

Množstvo informácií a prínosných diskusných príspevkov o generátoroch parserov, je možné nájsť na adrese [10].

3 Objektové technológie

3.1 Vytváranie objektov a garbage collector

S tvorbou a rušením objektov je spojená réžia, týkajúca sa správy pamäte. Oblasť pamäte pridelená objektom, sa v Jave označuje ako *halda*. Pri vytváraní objektu je nutné nájsť blok pamäte dostatočnej veľkosti, pri rušení objektu je nutné uvoľnený blok pamäte zaradiť do zoznamu voľných blokov. Programovací jazyk Java používa mechanizmus automatického uvoľňovania objektov z pamäte, takzvaný *garbage collector*. Hlavnou výhodou použitia garbage collectora je eliminácia častých chýb, ktoré vznikajú pri práci s pamäťou. Jedná sa predovšetkým o *dangling pointers* a *memory leaks*. K problému dangling pointers dochádza v situácii, keď je použitý ukazateľ, ktorý odkazuje na pamäť, ktorá už bola uvoľnená. K problému memory leaks dochádza v situácii, keď nie je uvoľnená pamäť, na ktorú už nevedú žiadne odkazy. Výhody, ktoré so sebou prináša garbage collector však nie sú zadarmo. Počas aktivity garbage collectora je chod aplikácie obmedzený, prípadne je celá aplikácia zastavená. To spôsobuje vyššie časové nároky aplikácie. Z dôvodu uvedenej réžie dochádza k uvoľňovaniu pamäte iba v momente zaplnenia haldy, keď nie je možné splniť požiadavku na alokáciu objektu. Komplexnejšie implementácie garbage collectora, so sebou zároveň prinášajú zvýšenú pamäťovú réžiu. S každou novou verziou jazyka Java, prichádzajú nové vylepšenia v implementácii garbage collectora, práve za účelom zníženia jeho celkovej réžie. Nasleduje popis jednotlivých implementácii garbage collectora, s hlavným zameraním na implementáciu v Jave 1.5.

3.1.1 Základné algoritmy

Hlavným problémom, ktorý musí riešiť každý garbage collector, je nájdenie rozdelenia objektov do dvoch skupín, na dosiahnuteľné a nedosiahnuteľné. Základná množina dosiahnuteľných objektov obsahuje objekty, na ktoré vedú odkazy zo statických premenných a z aktuálneho obsahu zásobníka. Celková množina dosiahnuteľných objektov, sa získa ako tranzitívny uzáver tejto množiny v relácií odkazov. Garbage collectory je možné rozdeliť do dvoch základných kategórií na:

- **Reference counting** - najjednoduchšia implementácia garbage collectora. S každým objektom je asociovaný počet odkazov na objekt. Každá operácia priradenia, prípadne návratu z funkcie, vyžaduje úpravu uvedeného čítača, pre všetky zainteresované objekty. V prípade, že hodnota čítača klesne na 0, je objekt označený ako nedosiahnuteľný a je zavolaná metóda na jeho uvoľnenie z pamäte. Implementácia garbage collectora prostredníctvom reference counting vyžaduje podporu zo strany prekladača, ktorý musí vygenerovať inštrukcie na zmenu čítačov a uvoľnenia objektu. Metóda reference counting nie je schopná pracovať s cyklickými štruktúrami.
- **Tracing collectors** - implementácia garbage collectora, ktorá je využívaná v programovacom jazyku Java. Garbage collector je zavolaný až v momente, keď je zaplnená halda a nie je možné splniť požiadavku na alokáciu objektu. Po aktivácii, zistí garbage collector množinu dosiahnuteľných objektov a zabezpečí uvoľnenie nedosiahnuteľných objektov.

V nasledujúcej časti budú popísané niektoré varianty tracing collectors garbage collectora.

Mark-sweep collectors

Garbage collector pracuje v dvoch fázach. V *mark fáze*, je označený každý dosiahnuteľný objekt. V *sweep fáze*, sa prechádzajú všetky objekty. V prípade, že objekt je nedosiahnuteľný je uvoľnený z pamäte a odpovedajúci blok pamäte je zaradený do zoznamu voľných blokov. Doba trvania celého procesu môže byť veľká a je závislá na veľkosti haldy. Garbage collector musí prechádzať všetky objekty, pritom v mnohých aplikáciách má drvivá väčšina objektov krátku životnosť. Ďalším problémom tejto metódy je nespojitosť voľných blokov pamäte, pretože tie môžu byť preložené využívanými blokmi. Tento problém sa označuje ako *fragmentácia haldy*. Pamäť dosiahnuteľných objektov je rozložená po celej oblasti haldy, čo znižuje pravdepodobnosť, že pri prístupe k objektu budú jeho dáta v cache procesora. Jedná sa o porušenie takzvaného *princípu lokality*.

Copying collectors

Halda je rozdelená na dve oblasti rovnakej veľkosti. Vždy je jedna aktívna a druhá neaktívna. Po zaplnení aktívnej oblasti, prichádza k slovu garbage collector. Zistí množinu dosiahnuteľných objektov a tie skopíruje na začiatok neaktívnej oblasti haldy. Úlohy oblastí sa potom vymenia. Táto metóda využíva zistenie uvedené v predchádzajúcej metóde, že väčšina objektov má krátku životnosť. Teda sa predpokladá, že po zaplnení oblasti, zostane len málo dosiahnuteľných objektov. Dosiahnuteľné objekty sa hromadia na začiatku oblasti a teda odpadá problém s fragmentáciou haldy a efektivitou prístupu k pamäti. Takisto cena alokácie objektov je pri použití tejto metódy minimálna. Objektu sa proste vyhradí pamäť na konci využitej časti oblasti. V prípade výkskytu veľkého počtu objektov s dlhou životnosťou, je však táto metóda neefektívna, pretože dochádza k opakovanému kopírovaniu objektov medzi oblasťami. Ďalšou nevýhodou tejto metódy je dvojnásobná veľkosť haldy.

Mark-compact collectors

Táto metóda v sebe kombinuje obe predchádzajúce metódy. Ako v prípade mark-sweep collectors sa jedná o dvojfázový proces. V prvej fáze sú označené dosiahnuteľné objekty. V druhej fáze sa znova prechádzajú všetky objekty. Dosiahnuteľné objekty sú kopírované na začiatok haldy. Podobne ako v prípade copying collectors, nedochádza k fragmentácii pamäte a dosiahnuteľné objekty sú pohromade. Objekty s dlhou životnosťou sa kumulujú na začiatku haldy a tak nedochádza k ich neustálemu kopírovaniu pri každej aktivite garbage collectora. Takisto nie je potrebné mať haldu dvojnásobnej veľkosti. Za tieto výhody oproti copying collectors, platí táto metóda nutnosťou prechádzať všetky objekty.

3.1.2 Generational collection

Ako už bolo spomenuté vyššie, každá z implementácií garbage collectora je vhodná pre iný typ objektov z pohľadu životnosti. *Generational collection* rozdeľuje haldu na dve oblasti, na *young generation* a *old generation*. Pre každú generáciu objektov môže byť použitá iná implementácia garbage collectora. Pri vytvorení je objekt umiestnený do young generation. V momente zaplnenia young generation, garbage collector vykoná takzvanú *minor collection*. Volanie minor collection vykoná čistenie pamäte iba v young collection. Pre young collection je typicky použitá copying collectors implementácia garbage collectora. Toto čistenie pamäte je teda veľmi rýchle a podľa predpokladov o životnosti objektov je väčšina z nich v momente volania garbage collectora nedosiahnuteľná. Objekty ktoré prežili určitý počet volaní garbage collectora, sú v minor collection presunuté do old

generation. V old generation sa tak hromadia objekty s dlhou životnosťou. V prípade neúspechu minor collection, buď z dôvodu uvoľnenia malého množstva pamäte alebo nedostatku miesta na kopírovanie objektov v old generation, sa zavolá takzvaná *major collection*. Major collection používa mark-compact collectors implementáciu garbage collectora a čistenie pamäte sa vykoná pre celú haldu. Po skončení major collection, sú všetky dosiahnuteľné objekty umiestnené na začiatku old generation. Major collection je výrazne pomalšia ako minor collection.

Rozdelenie haldy na dve oblasti umožňuje v minor collection obmedziť proces zisťovania dosiahnuteľných objektov iba na objekty v young generation. Teda odkazy vedúce do old generation, nemusia byť prechádzané. Problémom sú však odkazy z old generation do young generation, takzvané *old-to-young references*. Objekty v young collection, ktoré sú odkazované z dosiahnuteľných objektov z old generation, musia byť v minor collection takisto brané do úvahy. Old-to-young references môžu vzniknúť dvoma spôsobmi. Buď je v objekte, ktorý sa nachádza v old generation, zmenená referencia na objekt, ktorý sa nachádza v young generation alebo objekt v young generation, ktorý odkazuje na iný objekt v young generation, je presunutý do old generation. Existujú rôzne heuristiky na zisťovanie týchto odkazov, bez nutnosti prechádzať celý graf odkazov.

3.1.3 Ostatné metódy

V Jave 1.5 existujú okrem vyššie spomenutej tri ďalšie implementácie garbage collectora: *throughput collector*, *concurrent collector* a *incremental collector*. Tieto nové implementácie prinášajú výhody predovšetkým pre multiprocesorové systémy, kde je umožnený paralelný beh aplikácie a garbage collectora, prípadne súčasný beh garbage collectora na viacerých procesoroch. Ďalšie informácie o implementácii garbage collectora v Jave 1.5 je možné nájsť v článku [1].

3.2 Efektívne techniky na prácu s objektmi

3.2.1 Nemenné objekty

Nemenné objekty sú objekty, ktoré nemenia svoj stav. Typickým príkladom nemenných objektov sú inštancie triedy `String`. Nevýhodou nemenných objektov, je nutnosť vytvorenia novej inštancie po každej zmene stavu objektu. Teda každá metóda, ktorá modifikuje reťazec inštancie triedy `String`, vytvára novú inštanciu. Vďaka nemožnosti zmeniť stav objektu, odpadá problém s viacvláknovým prístupom k inštancii. Ďalšou výhodou použitia nemenných objektov je nenáročné klonovanie objektov. Pri klonovaní nemenného objektu, odpadá nutnosť vytvárať kópie objektov, ktoré sú odkazované z nemenného objektu. Jedná sa o takzvané *plytké kópie*. Typickým príkladom je metóda `substring`. Namiesto vytvorenia explicitnej kópie poľa znakov, v ktorom sú uložené znakové dáta reťazca, zdieľa nová inštancia triedy `String` znakové dáta s pôvodnou inštanciou. Za zmienku stojí ešte jedna vlastnosť nemenných objektov. Keďže nie je možné zmeniť v rámci objektu odkazy na iné objekty, nemôže z programového kódu dôjsť k vzniku problému old-to-young references, čo môže urýchliť činnosť garbage collectora.

3.2.2 Object pooling

Object pooling je technika, pri ktorej je rezervovaná pamäť, takzvaný *pool*, kde sa ukladajú nepoužívané objekty. Po skončení práce s objektom je objekt umiestený do poolu. V prípade potreby vytvorenia objektu, sa objekt získa z poolu. Keď v poole nie sú žiadne

objekty k dispozícii, vytvorí sa nový. Použitie poolu eliminuje asistenciu garbage collectora pri rušení objektov. Pri získaní objektu z poolu, nie je nutné vykonávať jeho kompletnú inicializáciu ako v prípade volania konštruktora. Nevýhodou použitia tejto techniky je nutnosť explicitného vracania objektov do poolu a zvýšené pamäťové nároky aplikácie. Explicitným vracaním objektov do poolu môže vzniknúť problém dangling pointers, keď existuje ukazateľ na objekt, ktorý už bol uvoľnený. Príklad implementácie object poolingu je možné nájsť v článku [2], kritiku tejto techniky v článku [3].

3.2.3 Constant pool

Technika používaná pre nemenné objekty. Často používané objekty sú umiestnené do poolu. V prípade potreby vytvorenia objektu, ktorý spĺňa kritéria nejakého objektu z poolu, sa objekt získa z poolu. Constant pool je napríklad použitý v triedach `String` a `Integer`.

Trieda String

Do poolu sa ukladajú všetky konštantné reťazce. Napríklad pre zdrojový kód:

```
String s1 = "abc"
String s2 = "abc"
```

budú obe premenné odkazovať na rovnakú inštanciu triedy `String`.

Trieda Integer

Constant pool je použitý pre často používané čísla v rozmedzí -128 až 127. Je vytvorené pole objektov reprezentujúcich tieto čísla. Objekt sa získava priamym prístupom do poľa. Túto implementáciu je možné nájsť v zdrojových súboroch Javy, ktoré sú súčasťou prostredia Java SE Development Kit (JDK) 5.0. Konkrétne sa jedná o metódu `valueOf`, definovanú v triede `java.lang.Integer`.

3.3 Kolekcie

Jedná sa o dátové štruktúry bežne používané v programoch, ako napríklad zásobník, fronta, hašovacia tabuľka atď. Tieto dátové štruktúry sú implementované v štandardných knižniciach programovacieho jazyka Java, čo urýchľuje prácu programátora. Implementácia je vytvorená na všeobecnej úrovni, aby tak boli pokryté všetky možné požiadavky pri používaní kolekcií.

Typ položiek

Jedna zo základných vecí, ktorých sa všeobecnosť týka je dátový typ položiek uchovávaných v kolekcii. V Jave sú položky reprezentované generickým typom `Object`, na ktorý je pretypovateľný ľubovoľný objektový typ. Nevýhodou tohto riešenia, je nutnosť explicitného pretypovania na požadovaný typ, pri každom získavaní položky z kolekcie. To môže viesť k vzniku chýb, ktoré sa ťažko odhaľujú, keďže typová kontrola sa vykonáva až počas behu aplikácie. Ďalšou nevýhodou je nemožnosť vytvoriť kolekciu zloženú z položiek primitívneho dátového typu.

V programovacom jazyku C++ je problém všeobecnosti dátových štruktúr riešený prostredníctvom šablón. Šablóna predstavuje implementáciu dátovej štruktúry bez

definovania typu položiek. Pri vytváraní inštancie šablóny programátor určí typ uchovávaných položiek a prekladač automaticky vygeneruje kód implementácie dátovej štruktúry s definovaným typom. Použitím šablón nevzniká problém s explicitným pretypovaním a je možné používať primitívne dátové typy. Nevýhodou tohto riešenia, je nutnosť vygenerovať kód implementácie dátovej štruktúry pre každý použitý typ.

Od verzie 1.5 bola do Javy pridaná podpora *parametrických typov*. Tie umožňujú vytvárať inštancie kolekcií so špecifickým typom. Oproti šablónam z C++, sa však negeneruje kód implementujúci kolekciu, ale je automaticky vygenerovaný kód na pretypovanie na definovaný typ. Na jednej strane tak nie je nutné generovať kód implementácie kolekcie pre každý použitý typ zvlášť, na strane druhej je však stále nutné pretypovanie na požadovaný typ a nie je možné vytvoriť inštanciu kolekcie obsahujúcu položky primitívneho dátového typu.

Viacvláknový prístup

Kolekcie sú implementované tak, aby umožňovali prístup k dátovým položkám inštancie z viacerých vláken. V praxi to znamená, že každá metóda kolekcie je definovaná s kľúčovým slovom `synchronized`.

Defenzívna implementácia metód

Metódy pred vykonaním svojej činnosti kontrolujú korektnosť vstupných parametrov. Jedná sa napríklad o kontrolu prístupového indexu voči rozsahu poľa atď.

4 Analýza vlastností objektových technológií

4.1 Testy vytvárania a rušenia objektov

Cieľom testov bola lexikálna analýza HTML dát o veľkosti 5,5 GB, rozdelených do 222 súborov. Veľkosť súborov sa pohybovala od 17,8MB do 32,5MB. Lexikálny analyzátor bol vytvorený pomocou nástroja Achilex 1.1.3, ktorý bol implementovaný v rámci diplomovej práce. Hlavná metóda testov, volá v cykle metódu na získanie tokena. V prípade, že je vrátený token reprezentujúci koniec súboru, je lexikálny analyzátor reštartovaný a je pustená lexikálna analýza ďalšieho súboru. Počas behu neboli vytvárané a rušené žiadne iné objekty okrem testovaných. Výnimkou sú objekty použité v štandardných Java knižniciach na čítanie vstupných dát. Frekvencia ich vytvárania a rušenia je však oproti testovaným objektom zanedbateľná. Nasleduje prehľad testov, vrátane dosiahnutých výsledkov.

4.1.1 Testy réžie garbage collectora

V každom teste bolo vygenerovaných 1 101 547 012 tokenov. Boli porovnávané dva parametre:

Parameter 1 – Životnosť objektov

Boli porovnávané nasledujúce hodnoty parametra:

- **Short-lived** - Jedná sa o test objektov s krátkou životnosťou. Po vrátení tokena z lexikálneho analyzátora, sa otestuje jeho identifikátor voči identifikátoru konca súboru. Po vykonaní tejto akcie, nie je viac objekt tokena potrebný.
- **Long-lived** - Jedná sa o test objektov s dlhou životnosťou. Po otestovaní identifikátora tokena voči identifikátoru konca súboru, sa token vloží do poľa o dĺžke 10 000 prvkov. Pole sa plní zľava doprava. Po zaplnení sa začne pole plniť znova od začiatku.

Parameter 2 – Implementácia tokena

Token je implementovaný pomocou nasledujúcej triedy:

```
public class Token {
  public int beginLine;
  public int beginColumn;
  public int endLine;
  public int endColumn;
  public int charCount;
  public int tokenId;
}
```

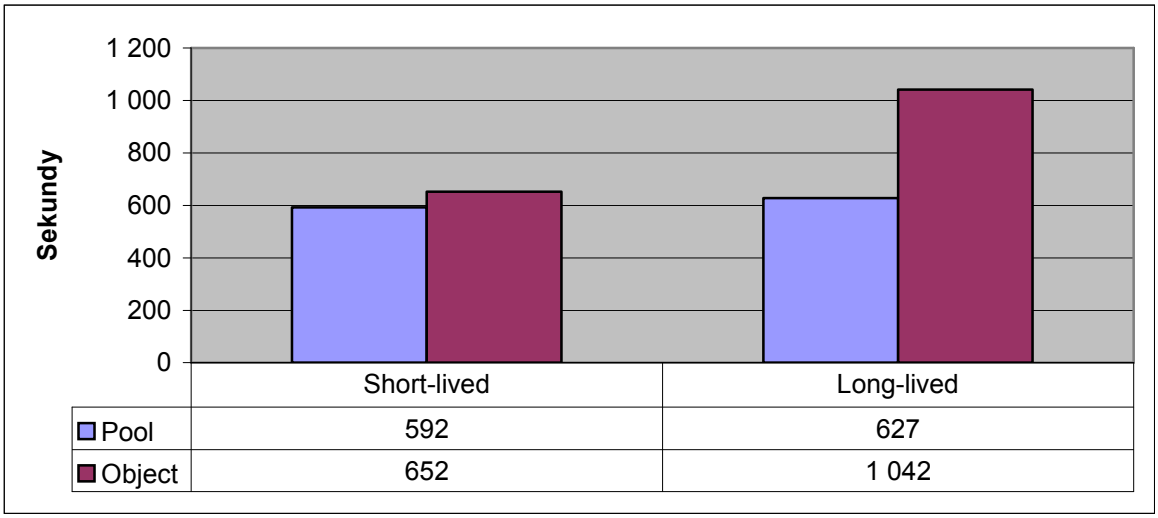
Atribúty `beginLine`, `beginColumn`, `endLine` a `endColumn` reprezentujú pozíciu tokena vo vstupnom súbore. Atribút `charCount` predstavuje počet znakov od začiatku súboru, kde token začína. V atribúte `tokenId` je uložený identifikátor tokena.

Boli porovnávané nasledujúce implementácie tokena:

- **Pool** - V tejto variante je využitá technika znovu využitia nepoužívaných objektov, takzvaný object pooling, ktorá bola popísaná v predchádzajúcej kapitole. Pre test objektov s krátkou životnosťou, je znovu využívaná jediná inštancia triedy tokena. Pre test objektov s dlhou životnosťou, je opakovane využívaná množina 10 001 inštancií. Odkazy na poolované inštancie sú udržiavané priamo v použitom poli. Inštancie pre tokeny sú získavané priamym prístupom do tohto poľa.
- **Object** - Pri generovaní tokena, sa vždy vytvára nová inštancia.

Vyhodnotenie výsledkov

Kombináciou vyššie uvedených hodnôt parametrov boli vytvorené 4 testy. Časy potrebné na jednotlivé testy dokumentuje nasledujúci graf:



Graf 1: Porovnanie časov pre jednotlivé testy.

V nasledujúcich tabuľkách je zaznamenaná aktivita garbage collectora pre jednotlivé testy. Na získanie informácii o činnosti garbage collectora, boli testy pustené s parametrom `verbose:gc`. V stĺpcoch tabuliek sú uvedené jednotlivé testy. Názov pred lomítkom označuje hodnotu druhého parametra a názov za lomítkom hodnotu prvého parametra. V riadkoch tabuliek, údaj *Minor GC* informuje o vykonaní minor collection a údaj *Major GC* o vykonaní major collection. V Tabuľke 1 sú uvedené časy strávené v minor a major collections garbage collectora. V Tabuľke 2 sú uvedené počty minor a major collections.

	Pool/Short	**Object/Short**	**Pool/Long**	**Object/Long**
Minor GC	0,007	8,711	0,011	130,631
Major GC	0,000	0,019	0,000	269,086
Súčet	0,007	8,730	0,011	399,711

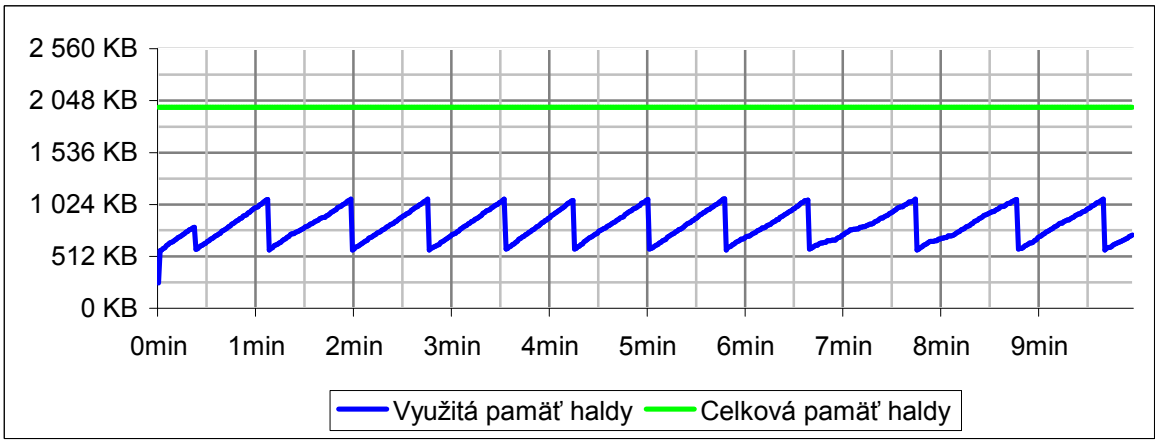
Tabuľka 1: Doba trvania behu garbage collectora v sekundách.

	Pool/Short	**Object/Short**	**Pool/Long**	**Object/Long**
Minor GC	13	67 493	13	67 345
Major GC	0	2	0	22 447
Súčet	13	67 495	13	89 792

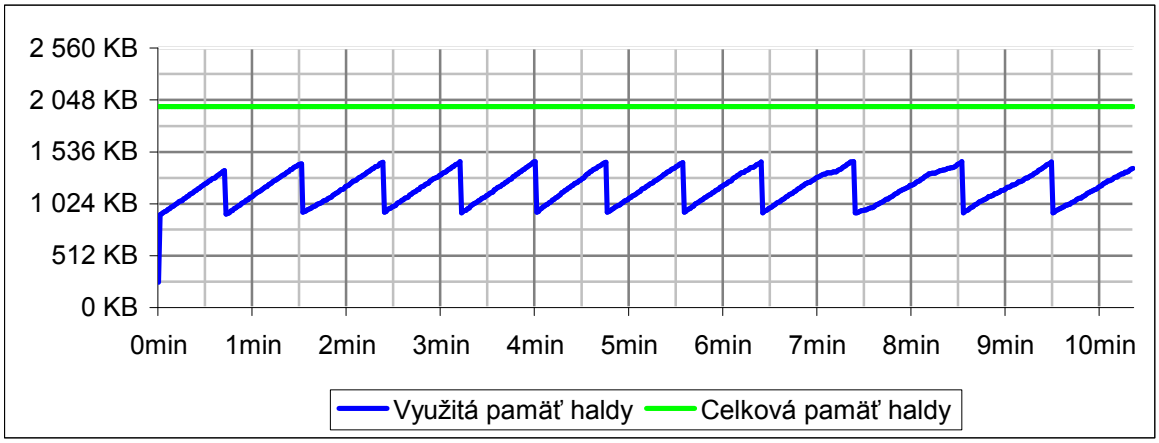
Tabuľka 2: Počty volaní garbage collectora.

Porovnanie 1 – Testy Pool/Short a Pool/Long

Testy *Pool/Short* a *Pool/Long*, dopadli podľa očakávaní najlepšie. Z tabuliek je vidieť minimálnu aktivitu garbage collectora. Na nasledujúcich grafoch je zobrazené využitie pamäte pre tieto testy.



Graf 2: Využitie pamäte pre test Pool/Short.

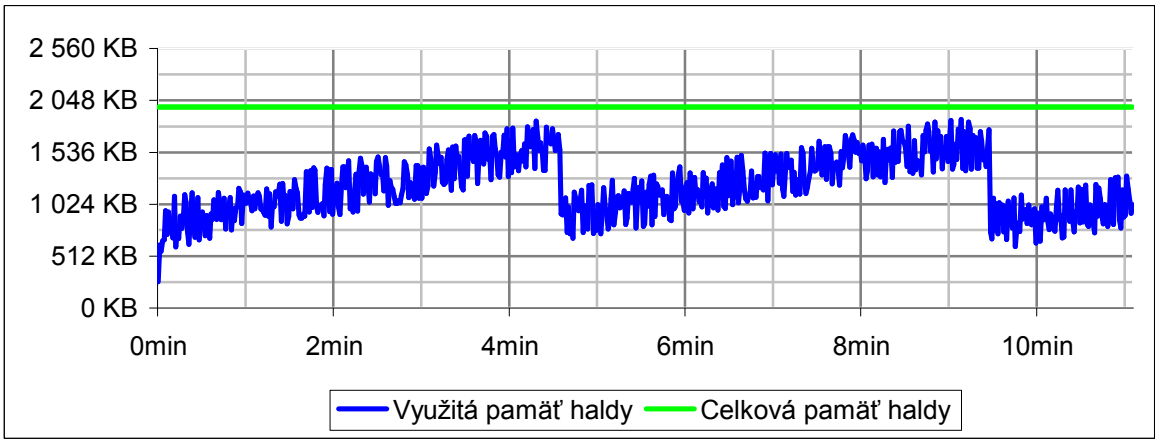


Graf 3: Využitie pamäte pre test Pool/Long.

Ako je vidieť z grafov, je využitie pamäte stabilné. Občasný nárast a uvoľnenie pamäte, je spôsobené alokáciou objektov v štandardných knižniciach na čítanie vstupných dát. Dôvodom vyššieho využitia pamäte v teste *Pool/Long* oproti testu *Pool/Short*, je väčší počet objektov, s ktorými sa v teste *Pool/Long* pracuje, vrátane poľa, ktoré objekty uchováva.

Porovnanie 2 – Test Object/Short

Ako je vidieť z Tabuľky 2, došlo v teste k vysokému počtu minor collections a dvom major collections. Pri pohľade na Tabuľku 1, je však viditeľné, že čas strávený v garbage collectore je malý. Je to približne 1,30% z celkovej doby behu testu. Pri porovnaní doby behu testov *Pool/Short* a *Object/Short*, je test *Pool/Short* rýchlejší o 10,10%. Na nasledujúcom grafe je znázornené využitie pamäte pre test *Object/Short*.

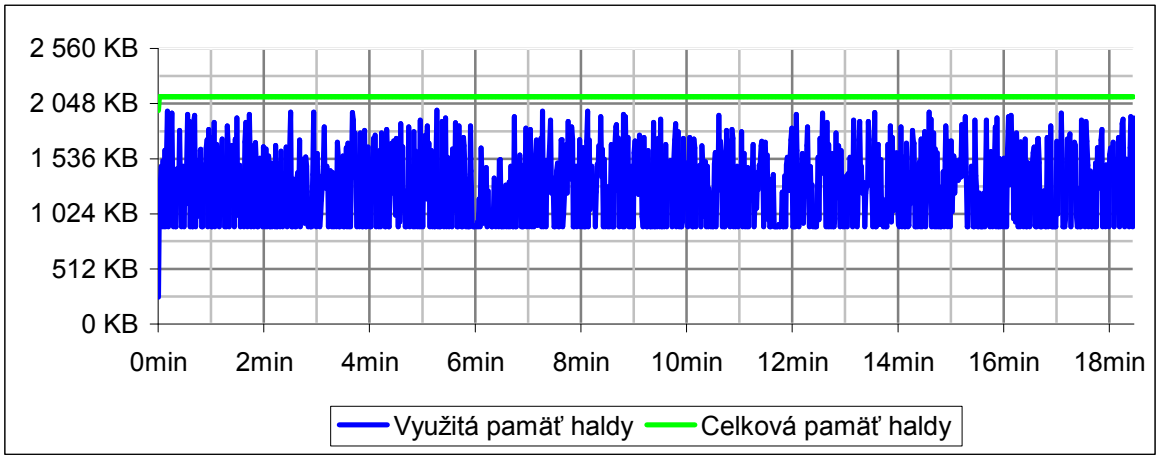


Graf 4: Využitie pamäte pre test Object/Short.

Z grafu je vidieť časté mierne nárasty a poklesy vo využití pamäte, spôsobené frekventovaným vytváraním objektov a následným volaním minor collections. Dvakrát došlo k zaplneniu celej haldy, v dôsledku čoho bola zavolaná major collection a došlo k výraznému uvoľneniu pamäte.

Porovnanie 3 – Test Object/Long

Z Tabuliek 1 a 2 je vidieť výraznú aktivitu garbage collectora. Počet minor collections je podobný ako v teste *Object/Short*, avšak čas strávený v minor collections je výrazne vyšší. Počet major collections je približne 3x menší ako počet minor collections, avšak čas strávený v major collections je 2x vyšší. Čas strávený v garbage collectore predstavuje 38,36% z celkovej doby behu testu. Pri porovnaní doby behu testov *Pool/Long* a *Object/Long*, je test *Pool/Long* rýchlejší o 66,19%. Na nasledujúcom grafe je znázornené využitie pamäte pre test *Object/Long*.



Graf 5: Využitie pamäte pre test Object/Long.

Z grafu je vidieť, že došlo k zvýšeniu celkovej veľkosti haldy. Dochádza často k zaplneniu haldy, čo je spôsobené kopírovaním živých objektov do old generation pri minor collection. Po zaplnení haldy je zavolaná major collection a je uvoľnená veľká časť pamäte.

4.1.2 Testy vytvárania reťazcov

V týchto testoch token uchováva informácie o reťazci, ktorý reprezentuje lexém. Jedná sa o testy vytvárania objektov s krátkou životnosťou. Rozdiely sú v implementácii reťazca. V každom teste bolo vygenerovaných 1 101 542 090 tokenov. Celková veľkosť znakových dát reťazcov bola 10,8GB. Priemerná veľkosť znakových dát jedného reťazca bola 10,5 bytov. Token je implementovaný pomocou nasledujúcej triedy:

```
public class Token {
  public char[] value;
  public int beginIndex;
  public int endIndex;
  public int hash;
}
```

Dátová veľkosť inštancií tokena, bez poľa obsahujúceho znaky, je rovnaká ako veľkosť inštancií triedy `String`. Atribút `value` uchováva odkaz na znakové pole reťazca. Atribúty `beginIndex` a `endIndex` reprezentujú umiestnenie znakových dát reťazca v znakovom poli. Atribút `hash`, slúži na uloženie hašovacieho kódu, ktorý však v testoch nebol využitý. Koniec súboru je identifikovaný vrátením tokena s hodnotou `null`.

Pool

V tomto teste je znovu využívaná jediná inštancia tokena. Nedochádza k vytváraniu znakového poľa reťazca, ale je predaný odkaz na znakové dáta vo vstupnom buffery lexikálneho analyzátora.

No-array

Pre každý token je vytvorený nový objekt a podobne ako v predchádzajúcom prípade, nedochádza k alokácii znakového poľa reťazca.

Empty-array

V tomto teste dochádza k vytváraniu znakového poľa reťazca, ale znakové dáta do neho nie sú kopírované. Konštruktor triedy má nasledujúci tvar:

```
public Token(int beginIndex, int endIndex) {
  int count = endIndex - beginIndex;
  this.value = new char[count];
  this.beginIndex = 0;
  this.endIndex = count;
}
```

Array

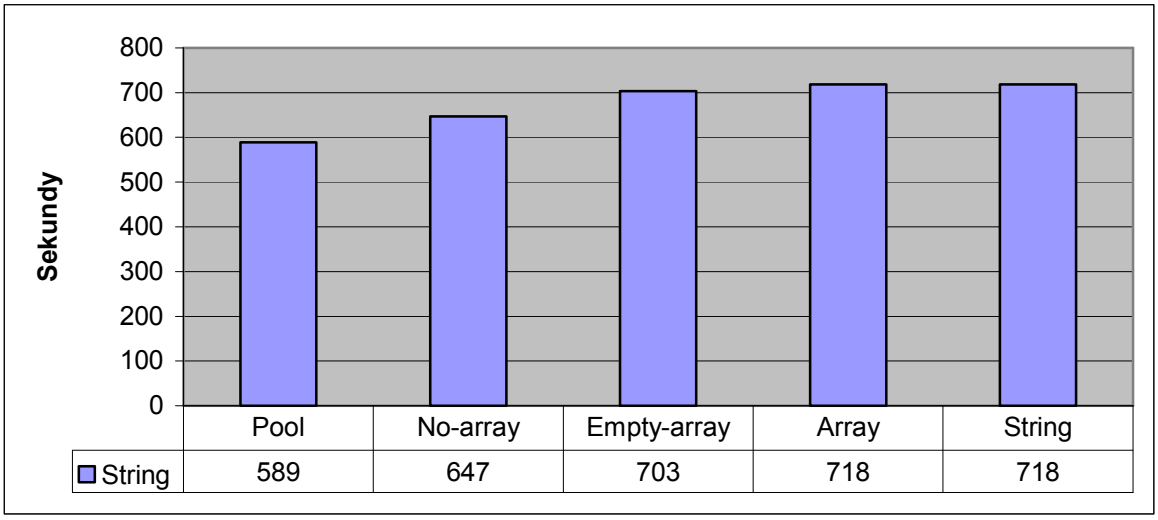
Oproti predchádzajúcemu testu je v konštruktore tokena vytvorená kópia znakových dát tokena:

```
public Token(char[] value, int beginIndex, int endIndex) {
  int count = endIndex - beginIndex;
  char[] v = new char[count];
  System.arraycopy(value, beginIndex, v, 0, count);
  this.value = v;
  this.beginIndex = 0;
  this.endIndex = count;
}
```

String

V tomto teste je token implementovaný pomocou triedy `String`.

Vyhodnotenie výsledkov



Graf 6: Porovnanie časov pre jednotlivé testy.

Rozdiel medzi testami *No-array* a *Empty-Array* je spôsobený vytváraním znakových polí v teste *Empty-array*. V teste *Array*, je vidieť časovú réžiu spôsobenú kopírovaním znakových dát zo znakového poľa buffera. Časy pre testy *Array* a *String* sú rovnaké, pretože implementácia triedy `String` je ekvivalentná s implementáciou použitou v teste *Array*. Test *No-array* je o 10,97% rýchlejší ako test *String*. Časová zložitosť vytvárania inštancií triedy `String` je teda o niečo vyššia ako pri bežných objektoch.

4.2 Kolekcie

Cieľom testov bola lexikálna a syntaktická analýza HTML dát o veľkosti 5,5GB, rozdelených do 222 súborov. Veľkosť súborov sa pohybovala od 17,8MB do 32,5MB. Lexikálny analyzátor bol vytvorený pomocou nástroja JFlex 1.4.1. Na vytvorenie syntaktického analyzátora bol použitý nástroj CUP 11a beta 20060330. V syntaktickom analyzátore dochádzalo k zápisom textových elementov PCDATA do dočasného súboru. Po spracovaní HTML súboru, bol obsah dočasného súboru zahodený, došlo k reštartu lexikálneho a syntaktického analyzátora a pokračovalo sa so spracovaním ďalšieho HTML súboru. Porovnávané boli rôzne implementácie zásobníka syntaktického analyzátora. Syntaktický analyzátor používa 2 zásobníky, hlavný, v ktorom sa uchovávajú inštancie triedy `Symbol`, ktorá reprezentuje terminály a neterminály a zásobník používaný pri zotavovaní zo syntaktických chýb, ktorý uchováva inštancie triedy `Integer`. V nasledujúcej tabuľke sú uvedené počty volaní pre jednotlivé operácie nad zásobníkmi.

	push	**pop**	**peek**	**elementAt**
Stack	2 702 507 819	2 702 507 375	4 381 414 123	972
VStack	1 722	750	1 944	0

Tabuľka 3: Prehľad počtu jednotlivých operácií nad zásobníkmi.

Zásobník *Stack* reprezentuje hlavný zásobník syntaktického analyzátora, zásobník *Vstack* reprezentuje zásobník použitý na zotavenie z chýb. Operácie `push` a `pop` predstavujú vloženie prípadne odobranie objektu z vrcholu zásobníka, operácia `peek` predstavuje získanie objektu z vrcholu zásobníka a `elementAt` je operácia, ktorá získa objekt z určenej pozície na zásobníku. Nasleduje prehľad jednotlivých testov.

Old-Stack

Zásobníky syntaktického analyzátora sú implementované pomocou inštancií triedy `Stack`, ktorá sa nachádza v balíku `java.util`. Nie je špecifikovaný typ uchovávaných položiek, a preto je nutné explicitné pretypovávanie na požadovaný typ, pri získavaní položky zo zásobníka.

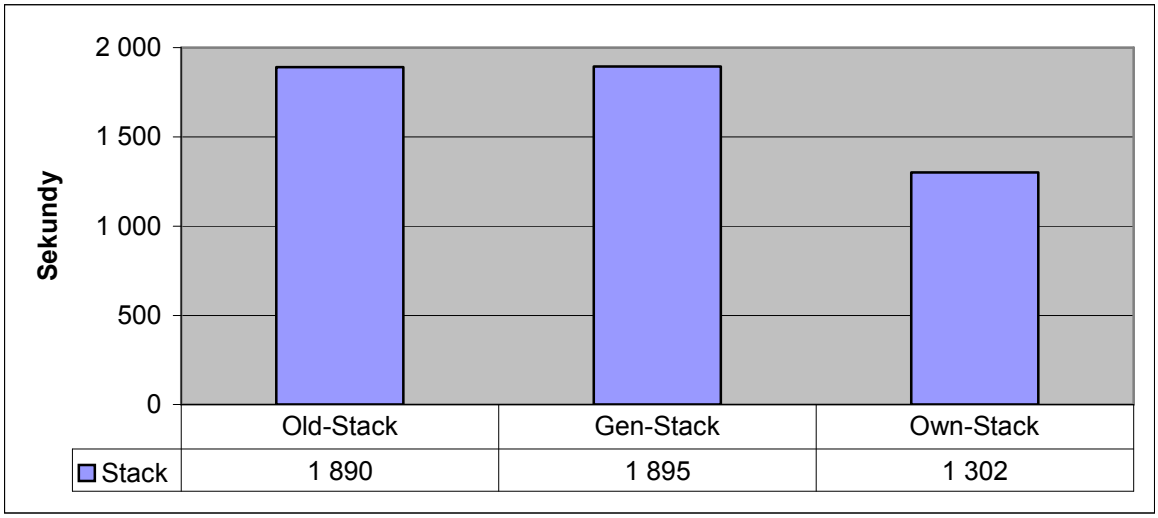
Gen-Stack

Oproti predchádzajúcemu testu je špecifikovaný typ uchovávaných položiek a teda nie je nutné explicitné pretypovávanie.

Own-Stack

Namiesto využitia štandardnej triedy jazyka Java, je zásobník implementovaný pomocou poľa špecifického typu. Operácie nad zásobníkom sú implementované priamym prístupom do poľa.

Vyhodnotenie výsledkov



Graf 7: Porovnanie časov pre jednotlivé testy.

Z grafu je vidieť, že použitie parametrických typov prináša predovšetkým zvýšený komfort pri programovaní. Z pohľadu doby behu, sú výsledky pre testy *Old-Stack* a *Gen-Stack* v podstate identické. Vlastná implementácia zásobníkov, priniesla výrazne zrýchlenie. Test *Own-Stack* je 1,5x rýchlejší ako test *Old-Stack*.

5 JavaCC

V tejto kapitole bude popísaná architektúra obľúbeného LL(k) generátora parserov pre jazyk Java. Jedná sa o voľne šíriteľný generátor parserov. V dobe písania diplomovej práce bola na stránkach projektu [4] k dispozícii na stiahnutie verzia 4.0.

5.1 Špecifikácia parsera

Špecifikácia lexikálneho a syntaktického analyzátora je integrovaná v jednom súbore. Syntax vstupného súboru je nasledujúca:

```
javacc_input ::= javacc_options
  "PARSER_BEGIN" "(" <IDENTIFIER> ")"
  java_compilation_unit
  "PARSER_END" "(" <IDENTIFIER> ")"
  ( production )*
  <EOF>
```

V sekcii `javacc_options`, je možné nastaviť základné parametre výsledného parsera. Nasleduje prehľad niektorých zaujímavých parametrov:

- **`LOOKAHEAD`** - Hodnotou parametra je číslo, ktoré určuje dĺžku výhľadu vo vygenerovanom parsery.
- **`UNICODE_INPUT`** - Nastavením tohto parametra na hodnotu `true`, je možné v parsery spracovávať znaky v rozmedzí 0 - 65 535.
- **`BUILD_PARSER`** - Nastavením tohto parametra na hodnotu `false`, je možné použiť JavaCC ako generátor lexikálnych analyzátorov. Rozhranie vygenerovaného analyzátora je pevne dané.
- **`USER_TOKEN_MANAGER`** - Tento parameter umožňuje použiť namiesto automaticky vygenerovaného, vlastný lexikálny analyzátor.
- **`JDK_VERSION`** - Implicitne je vygenerovaný zdrojový kód kompatibilný s Javou vo verzii 1.4. Pri pokuse o kompiláciu takéhoto zdrojového kódu pod Javou 1.5, dôjde k výpisu varovných hlášok ohľadom použitia kolekcií bez špecifického typu. JavaCC však umožňuje pri správnom nastavení tohto parametra, vygenerovanie zdrojového kódu kompatibilného s Javou 1.5.

Za kľúčovými slovami `PARSER_BEGIN` a `PARSER_END`, je uvedený názov triedy parsera. V sekcii *`java_compilation_unit`*, sa nachádza zdrojový kód triedy parsera. Do tohto kódu, sa skopíruje vygenerovaný kód parsera.

Sekcie `production` obsahujú bloky regulárnych výrazov a pravidiel gramatiky.

5.1.1 Blok regulárnych výrazov

Blok regulárnych výrazov má nasledujúcu syntax:

```
regular_expr_production ::= [ lexical_state_list ]
  regexpr_kind [ "[" "IGNORE_CASE" "]" ] ":"
  "{" regexpr_spec ( "|" regexpr_spec )* "}"

regexpr_spec ::= regular_expression
  [ java_block ]
  [ ":" java_identifier ]
```

V časti `lexical_state_list`, je uvedený zoznam stavov lexikálneho analyzátora. Aby bolo možné spustiť rozpoznávanie lexémov podľa ďalej uvedených regulárnych výrazov, lexikálny analyzátor sa musí nachádzať v jednom z uvedených stavov.

V časti `regexpr_kind`, je uvedený typ akcie, ktorý sa má vykonať po rozpoznaní lexému pre nejaký regulárny výraz. K dispozícii sú nasledujúce možnosti:

- **`SKIP`** - rozpoznaný lexém je zahodený a pokračuje sa v lexikálnej analýze.
- **`MORE`** - rozpoznaný lexém sa stane prefixom lexému rozpoznaného v nasledujúcej zhode. Použitím tejto akcie je tak možné skladať reťazec lexému z čiastkových reťazcov v jednotlivých zhodách.
- **`TOKEN`** - asi najpoužívanejšia možnosť. Po rozpoznaní lexému, sa vygeneruje token, ktorý je vrátený syntaktickému analyzátoru.
- **`SPECIAL_TOKEN`** - tak ako v predchádzajúcom prípade, je vygenerovaný token, ktorý je vrátený syntaktickému analyzátoru. Pre syntaktický analyzátor je však tento token neviditeľný, ale je možné k nemu pristupovať v akciách syntaktického analyzátora.

Po špecifikácii typu akcie, nasleduje zoznam regulárnych výrazov. Ku každému regulárnemu výrazu môže byť určený identifikátor generovaného tokena, lexikálna akcia (sekcia *`java_block`*) a stav, do ktorého po zhode lexikálny analyzátor prejde (sekcia *`java_identifier`*).

Na nasledujúcom príklade je uvedený blok s jediným regulárnym výrazom. Po rozpoznaní identifikátora zloženého z písmen a čísel, sa vykoná jeho konverzia na veľké písmená. Syntaktickému analyzátoru je vrátený token s identifikátorom IDENTIFIER. Tento príklad je prebraný z gramatiky JavaCC. Doplnená bola iba akcia.

```
TOKEN :
{
  < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
  {
    matchedToken.image =
    matchedToken.image.toUpperCase();
  }
}
```

Premenná `matchedToken` reprezentuje vygenerovaný token pre danú zhodu. O premenných prístupných v akcii, bude reč v ďalšej časti.

5.1.2 Blok pravidiel gramatiky

Blok pravidiel gramatiky má nasledujúcu syntax:

```
bnf_production ::= java_access_modifier
  java_return_type
  java_identifier "(" java_parameter_list ")" ":"
  java_block
  "{" expansion_choices "}"
```

V jednom bloku sa nachádza množina pravidiel s rovnakým neterminálom na ľavej strane. Je vidieť, že gramatika nie je zapísaná pomocou klasických prepisovacích pravidiel, ale je využitá forma zápisu s deklaráciou metód, ktorá odráža výslednú podobu syntaktického analyzátora v zdrojovom kóde, tak ako to bolo popísané v kapitole 2. Teda sa dá povedať, že jeden blok obsahuje práve jednu metódu, ktorá reprezentuje neterminál. Blok začína uvedením hlavičky metódy. V sekcii `java_block`, môže byť umiestnený zdrojový kód, ktorý sa skopíruje na začiatok tela metódy. V tejto časti bývajú umiestnené deklarácie premenných, používaných v akciách. V sekcii `expansion_choices`, sú umiestnené telá pravidiel v EBNF notácii a akcie definované programátorom. Nasleduje jednoduchý príklad:

```
StateList state_list() :
{
  Token t;
  StateList slist = new StateList();
}
{
  LOOKAHEAD(2)
  "<" "*" ">"
  {slist.setAll(); return slist;}
  |
  "<"
  t=<IDENTIFIER>       {slist.addState(t.image);}
  ( "," t=<IDENTIFIER> {slist.addState(t.image);} )*
  ">"
  {return slist;}
}
```

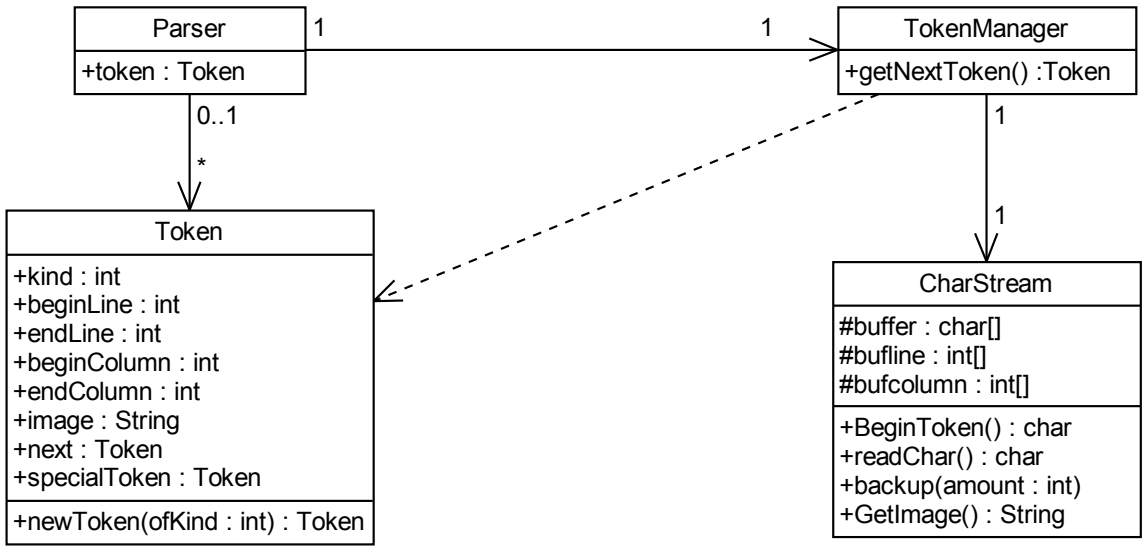
Príklad je prebratý z gramatiky JavaCC. Slúži na rozpoznanie prípustných stavov v definícii bloku regulárnych výrazov. Prvé pravidlo analyzuje zápis, ktorý pripúšťa všetky stavy. Druhé pravidlo analyzuje zápis, v ktorom je uvedený zoznam prípustných stavov. Pri rozhodovaní, ktoré z týchto dvoch pravidiel použiť, je nutné poznať prvé dva tokeny. Preto je dĺžka výhľadu nastavená na 2. Pomocou konštrukcie `LOOKAHEAD`, je možné definovať aj syntaktické a sémantické predikáty. Ako je vidieť, v pravidlách je možné použiť aj reťazcové konštanty. Generátor z nich automaticky vytvorí definície tokenov.

Na príklade je vidieť spôsob, ktorým sa pristupuje k tokenu v syntaktickom analyzátore. Referencia na token je predaná lokálnej premennej `t`. V akcii dochádza k pridaniu reťazca identifikátora do kolekcie `StateList`, ktorej inštancia je vytvorená na začiatku tela metódy. Po spracovaní stavov je metóda ukončená, vrátením vytvorenej

inštancie triedy `StateList` volajúcej metóde.

5.2 Analýza vygenerovaného parsera

V tejto časti bude popísaná architektúra vygenerovaného parsera, so zameraním na efektivitu implementácie. Výstupom generátora je sada tried, kde každá trieda rieši špecifickú úlohu parsera.



Obrázok 1: Diagram tried vygenerovaného parsera.

5.2.1 Trieda TokenManager

Trieda `TokenManager` implementuje lexikálny analyzátor. Triede `Parser` poskytuje metódu `getNextToken`, ktorá sa stará o generovanie tokenov. Ku generovaniu tokenov dochádza automaticky v blokoch regulárnych výrazov, ktoré majú definovaný typ akcie ako `TOKEN` prípadne `SPECIAL_TOKEN`. Vygenerovanie tokena je proces, pri ktorom dôjde k vytvoreniu inštancie triedy `Token`, a vyplneniu jeho atribútov.

Atribút `kind` obsahuje identifikátor tokena. Jedná sa o najdôležitejší atribút tokena, pomocou ktorého vykonáva syntaktický analyzátor porovnanie s terminálnymi symbolmi pravidiel gramatiky. Pre náš príklad, uvedený v predchádzajúcej časti, by atribút `kind` obsahoval hodnotu `IDENTIFIER`. Atribúty `beginLine`, `beginColumn`, `endLine` a `endColumn`, reprezentujú pozíciu tokena vo vstupnom súbore. Informácia o pozícii je dôležitá pri hlásení o výskyte chyby vo vstupných dátach, keď je potrebné lokalizovať miesto chyby. Atribút `image` reprezentuje lexém, teda reťazec rozpoznaný v zhode. Inštancia reťazca sa získa volaním metódy `GetImage` triedy `CharStream`. Táto metóda vytvorí inštanciu triedy `String` zo znakovej sekvencie, ktorá reprezentuje nájdený reťazec vo vstupnom buffery. V prípade, že regulárny výraz, podľa ktorého k zhode došlo, je definovaný ako konštantný reťazec, nevytvára sa nová inštancia triedy `String`, ale reťazec sa vezme z poľa konštantných reťazcov. Atribút `next` slúži na vytváranie spojového zoznamu tokenov, ktorý je potrebný pre syntaktický analyzátor. Atribút `specialToken` obsahuje odkaz na token, ktorý vznikol v bloku definovanom ako `SPECIAL_TOKEN`. Ako už bolo spomínané, takéto tokeny sú syntaktickým analyzátorom ignorované a je k ním možné pristupovať práve pomocou tohto atribútu.

Teda pre každú zhodu v bloku označenom ako TOKEN, prípadne SPECIAL_TOKEN, dôjde automaticky k vytvoreniu inštancie triedy Token, spolu s vyplneným reťazcom. V akcii lexikálneho analyzátora, je možné pristupovať k tejto inštancii pomocou premennej matchedToken. V akciách umiestnených v blokoch označených ako TOKEN, SPECIAL_TOKEN, prípadne MORE, je k dispozícii premenná image, ktorá obsahuje odkaz na inštanciu triedy StringBuffer. Na začiatku akcie sa k tejto inštancii automaticky pripojí aktuálny reťazec tokena. Po vykonaní akcie umiestnenej v bloku TOKEN, SPECIAL_TOKEN, prípadne SKIP, je inštancia zahodená a pred vykonaním nasledujúcej akcie sa automaticky vytvorí nová. Pomocou premennej image, je možné v akciách vykonávať modifikácie reťazca tokena a pomocou metódy toString, vygenerovať modifikovaný reťazec, ktorým môže byť nahradený automaticky vytvorený reťazec tokena.

V prípade potreby uchovania ďalších informácii v inštanciách tokenov, je možne definovať nové triedy tokenov odvodené od základnej triedy Token. Do tela metódy newToken triedy Token, sa potom umiestni zdrojový kód, ktorý na základe identifikátoru tokena, vygeneruje inštanciu požadovanej triedy.

Programátorovi je teda poskytnuté rozhranie na vysokej úrovni. Nemusí sa starať o vlastnoručné vytváranie tokenov a reťazec tokena je automaticky k dispozícii, vrátane možnosti jeho modifikácie. Na druhej strane však dochádza k frekventovanému vytváraniu objektov. Niekedy nie je potrebné poznať reťazec tokena, ale stačí informácia o nájdení tokena s určitým identifikátorom. Takisto premenná image zostáva často nevyužitá. Výsledky testov z kapitoly 4 ukazujú, že vytváranie objektov, a predovšetkým reťazcov, prináša pozorovateľnú časovú réžiu aj v prípade krátkej životnosti objektov.

5.2.2 Trieda CharStream

Trieda CharStream poskytuje znakové dáta vstupu, lexikálnemu analyzátoru. Vstupné dáta sa načítavajú po blokoch do buffera, ktorý zároveň slúži ako úložište znakových dát práve rozpoznávaného lexému. Buffer je implementovaný ako znakové pole, kde odkaz na inštanciu buffera je umiestnený v atribúte buffer. Na začiatku rozpoznávania lexému, volá lexikálny analyzátor metódu BeginToken, ktorá okrem toho, že vráti aktuálny znak zo vstupu, označí v buffery pozíciu, od ktorej budú uchovávané znaky, až do momentu rozpoznania lexému. Počas spracovávania lexému, je na získanie ďalších znakov zo vstupu, volaná metóda readChar. Pri potrebe načítania nových dát, niekedy nie je možné prepísať obsah celého buffera, pretože sa tam už nachádzajú znakové dáta práve rozpoznávaného lexému. V JavaCC je tento problém riešený využitím cyklického buffera. Teda nové dáta sa načítajú za koniec rozpoznávaného lexému. V prípade, že sa narazí na koniec buffera, sú nové dáta načítané od začiatku buffera. Pri tomto riešení môže nastať situácia, keď znakové dáta rozpoznaného lexému nie sú súvislé. V takomto prípade je nutné v metóde GetImage, ktorá vytvára reťazec tokena, použiť operáciu zreťazenia týchto dvoch nesúvislých častí.

V prípade, že veľkosť rozpoznávaného lexému presiahne veľkosť buffera, je nutné vykonať expanziu buffera. Pri expanzii sa vytvorí nový, väčší buffer, skopírujú sa do neho znakové dáta zo starého buffera a starý buffer sa zahodí. JavaCC používa buffer o veľkosti 4 096 znakov. Pri expanzii dochádza ku konštantnému zväčšovaniu dĺžky buffera o 2 048 znakov.

Atribúty `bufline` a `bufcolumn` uchovávajú čísla riadkov a stĺpcov pre všetky znaky v buffery. K vytvoreniu záznamu o pozícii dôjde pri volaní metódy `BeginToken`, prípadne `getChar`. Uchovanie čísel riadkov a stĺpcov v poliach je potrebné v prípade, keď dôjde k vráteniu znakov späť do buffera. K tejto operácii môže dôjsť v procese rozpoznávania lexému, alebo v lexikálnej akcii, volaním metódy `backup`. Pri tejto operácii, sa aktuálny ukazateľ na posledný spracovávaný znak v buffery, posunie o *n* miest dozadu, kde *n* je parameter metódy `backup`. Pri tomto posune je potrebné správne prepočítanie začiatočnej prípadne koncovej pozície rozpoznávaného lexému. Táto informácia je prístupná v poliach uchovávajúcich pozície. Pomocou týchto polí, je takisto možné zistiť pozíciu ľubovoľného znaku rozpoznaného reťazca. Súčasťou procesu expanzie buffera, je aj expanzia týchto polí. Pre každý znak v buffery je teda potrebných ďalších 8 bytov na uchovanie informácie o pozícii. V prípade použitia väčšieho buffera, tak môže dôjsť k výraznejšiemu zvýšeniu využitej pamäte.

Na tomto mieste je vhodné spomenúť ešte jednu vlastnosť triedy `CharStream`. Po spracovaní vstupného súboru, je možne parser reinicializovať a začať spracovanie ďalšieho súboru. Pri reinicializácii dochádza k vráteniu buffera na pôvodnú veľkosť. Teda v prípade, že počas behu parsera, došlo k expanzii buffera, je tento buffer zahodený a vytvorený nový s pôvodnou veľkosťou. Toto chovanie môže viesť k expanziám buffera, ktoré sa opakujú pre každý spracovávaný súbor. Je možné zväčšiť pôvodnú veľkosť buffera, ale požadovanú hodnotu niekedy nie je možné dobre spočítať.

5.2.3 Trieda Parser

Trieda `Parser` implementuje syntaktický analyzátor. Je zložená zo sady metód, kde každá metóda reprezentuje jeden blok pravidiel. Okrem týchto metód je súčasťou syntaktického analyzátora sada podporných metód, ktoré zabezpečujú správu tokenov, rozpoznávanie výhľadu atď. Syntaktický analyzátor si uchováva spojový zoznam tokenov. Z tohto dôvodu je jedným z atribútov triedy `Token`, atribút `next`, ktorý odkazuje na nasledujúci token v spojovom zozname. Spojový zoznam slúži na uchovávanie výhľadu. Po rozhodnutí o výbere pravidla podľa výhľadu, je potrebné tokeny vo výhľade uchovať, kvôli spracovaniu pravidla a spusteniu definovaných akcií. Pri spracovaní pravidla, je odkaz na naposledy spracovaný token uchovávaný v atribúte `token`. V prípade potreby ďalšieho tokena, sa odkaz na aktuálny token v atribúte `token`, posunie na nasledujúci token v spojovom zozname. V prípade, že atribút `token` ukazuje na posledný prvok v spojovom zozname, je nový token získaný od lexikálneho analyzátora, volaním jeho metódy `getNextToken`. Nový token sa zaradí na koniec spojového zoznamu a odkaz na neho sa zapíše do atribútu `token`. Aktuálny vstup je možne modifikovať aj z akcií, volaním metódy `getNextToken`, ktorá je definovaná v triede `Parser` a automaticky posunie vstup na nasledujúci token.

5.2.4 Zotavenie z chýb

Počas behu parsera môže dôjsť k vzniku chyby buď v lexikálnej analýze, pri nájdení neznámeho znaku alebo v syntaktickej analýze, keď je rozpoznaný neznámy token. Pri zotavení z chyby sú dôležité dve veci. Jednak je potrebné podať chybovú správu, kde najdôležitejšou informáciou je miesto výskytu chyby vo vstupných dátach. Ako už bolo spomenuté, JavaCC umožňuje automatické počítanie riadkov a stĺpcov jednotlivých znakov zo vstupu, teda tieto informácie sú k dispozícii.

Ďalšou dôležitou vecou je zotavenie z chyby, teda dostanie sa do stavu, kedy je možné pokračovať v analýze. Zotavenie z lexikálnej chyby je väčšinou realizované ignorovaním neznámeho znaku, prípadne domyslením si chýbajúcich znakov. Zotavenie z chýb v syntaktickom analyzátore, je v JavaCC riešené pomocou výnimiek. Každá metóda reprezentujúca neterminál gramatiky, môže vyhodiť výnimku typu `ParseException`. V prípade, že výnimka nie je v tele metódy odchytená, je spracovanie metódy a teda tela pravidla automaticky prerušené, metóda ukončená a výnimka sa propaguje do volajúcej metódy. Keď chce programátor reagovať na vznik chyby, musí príslušné pravidlo alebo pravidlá v tele metódy obaliť do try-catch bloku a v catch bloku umiestniť kód ktorý zabezpečí zotavenie z chyby. Pri akcii zotavenia sa typicky zo vstupu čítajú tokeny, až kým sa nenarazí na token, na ktorom je možné sa zosynchronizovať a pokračovať ďalej v analýze.

```
void stm() :
{}
{
  try {
    If_stm() <SEMI>
    |
    while_stm() <SEMI>
  } catch (ParseException e) {
    error_skipto(SEMI);
  }
}
```

Na tomto príklade dôjde pri vzniku chyby k synchronizácii na symbole bodkočiarky.

6 JFlex/CUP

JFlex [5] a CUP [6] sú generátory lexikálnych a syntaktických analyzátorov, ktoré odpovedajú nástrojom Flex a Bison pre jazyk C. CUP je zástupca LALR(1) generátorov. Programy sú voľne dostupné k stiahnutiu na svojich domovských stránkach. V čase písania diplomovej práce bola pre JFlex k dispozícii verzia 1.4.1 a pre CUP verzia 0.11a beta 20060608.

6.1 Špecifikácia parsera

Špecifikácia parsera je rozdelená do dvoch súborov. V jednom sa nachádza špecifikácia lexikálneho analyzátora pre JFlex, v druhom špecifikácia syntaktického analyzátora pre CUP.

6.1.1 Špecifikácia lexikálneho analyzátora

Vstupný súbor pre JFlex má nasledujúcu syntax:

```
jflex_input ::= user_code
  "%%" options_and_declarations
  "%%" lexical_rules
```

V sekcii *user_code* je umiestnený zdrojový kód v jazyku Java, ktorý sa skopíruje pred triedu vygenerovaného lexikálneho analyzátora. V tejto časti býva typicky umiestnená deklarácia názvu balíku, prípadne zoznam importovaných balíkov.

V sekcii `options_and_declarations` sú umiestnené nastavenia parametrov generovaného lexikálneho analyzátora. Ďalej môže byť do tejto sekcie umiestnený zdrojový kód, ktorý sa skopíruje do tela triedy lexikálneho analyzátora. Nasleduje prehľad niektorých zaujímavých parametrov:

- **`%unicode`** - Po nastavení tohto parametra, bude lexikálny analyzátor schopný rozpoznávať znaky v rozmedzí 0 - 65 535.
- **`%type`** - Určuje typ návratovej hodnoty metódy na získavanie tokena.
- **`%function`** - Určuje názov metódy na získavanie tokena.
- **`%implements`** - Zoznam rozhraní, ktoré vygenerovaná trieda lexikálneho analyzátora implementuje.

Pomocou parametrov `%type`, `%function` a `%implements` je možné špecifikovať rozhranie vygenerovaného lexikálneho analyzátora a ten pripojiť k ľubovoľnému syntaktickému analyzátoru.

V sekcii `lexical_rules`, sú umiestnené samotné pravidlá lexikálneho analyzátora. Pravidlo má nasledujúcu syntax:

```
lexical_rules ::= rule+

rule ::= [state_list] ["^"] reg_exp [lookahead] action
  | [state_list] "<<EOF>>" action
  | state_group

state_group ::= state_list "{" rule+ "}"
```

Teda špecifikácia lexikálneho analyzátora sa skladá zo zoznamu pravidiel, prípadne zoznamu blokov pravidiel. Blok pravidiel je analógiou blokov regulárnych výrazov v JavaCC. Opäť, definícia bloku začína množinou prípustných stavov, za ktorou nasleduje zoznam regulárnych výrazov, spolu s akciami. Je možné do seba bloky vnorovať. Chýba definícia typu akcie, identifikátoru tokena a stavu, do ktorého po zhode lexikálny analyzátor prejde. Na prechody medzi stavmi, je k dispozícii metóda `yybegin`, ktorú je možné používať v akciách. Keďže rozhranie lexikálneho analyzátora nie je pevne dané, automatické generovanie tokenov, ako v JavaCC nie je možné. Definovanie typu akcie a identifikátoru tokena preto nie je potrebné, a vygenerovanie tokena je v rukách programátora. Príklad lexikálneho pravidla z predchádzajúcej kapitoly, by v JFlex-e vyzeral takto:

```
{LETTER} ({LETTER}|{DIGIT})*
{
  return symbolFactory.newSymbol(
    "IDENTIFIER", IDENTIFIER,
    new Location(yyline + 1, yycolumn + 1),
    new Location(yyline + 1, yycolumn + yylength()),
    yytext().toUpperCase());
}
```

Metóda `newSymbol`, vytvorí inštanciu tokena podľa zadaných parametrov. Prvé dva parametre reprezentujú identifikátor tokena v podobe reťazca a v podobe čísla. Ďalšie dva parametre reprezentujú pozíciu tokena vo vstupnom súbore. Premenné `yyline` a `yycolumn` obsahujú číslo riadka, prípadne stĺpca, na ktorom reťazec tokena začína. Metóda `yytext` vygeneruje inštanciu triedy `String`, ktorá obsahuje znakové dáta rozpoznaného lexému.

6.1.2 Špecifikácia syntaktického analyzátora

Vstupný súbor pre CUP má nasledujúcu syntax:

```
cup_input ::= package_spec import_list
  code_parts
  (symbol)+ precedence_list start_spec
  (production)+
```

V sekcii `package_spec` je umiestnený názov balíku, v ktorom sa bude trieda vygenerovaného syntaktického analyzátora nachádzať. V sekcii `import_list`, je uvedený zoznam importovaných balíkov. Do sekcie `code_part`, je možné umiestniť zdrojový kód v jazyku Java, ktorý sa skopíruje do tela vygenerovanej triedy.

Ďalej nasleduje samotná špecifikácia syntaktického analyzátora. Najprv je uvedená deklarácia symbolov použitých v gramatike. Gramatika definuje dva druhy symbolov: terminály a neterminály. Deklarácia má nasledujúcu syntax:

```
symbol ::= ("terminal" | "non terminal")
  [ type_id ] symbol_names
```

Po uvedení druhu, je možné určiť typ hodnoty deklarovaných symbolov, za ktorým nasleduje zoznam názvov symbolov. Samotné pravidlá gramatiky majú nasledujúcu syntax:

```
production ::= non_terminal "::=" expansion_choices
```

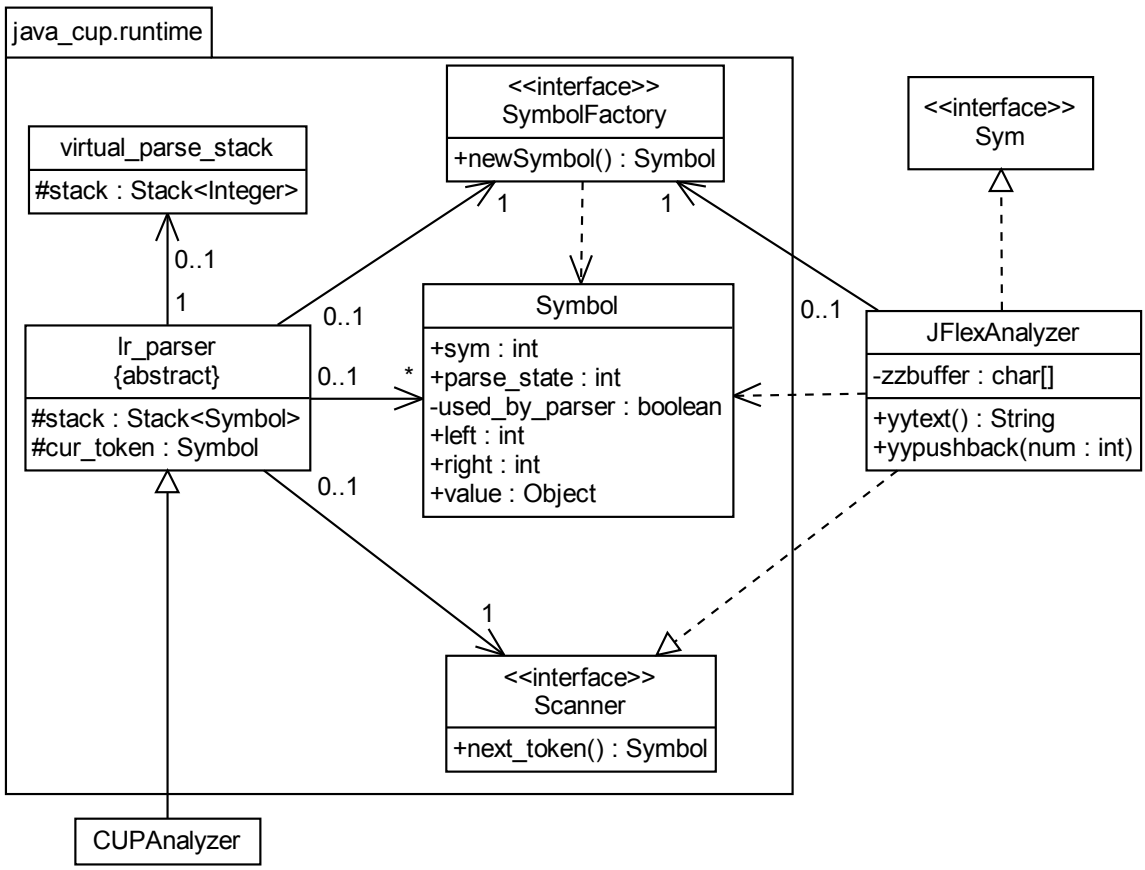
Jedná sa teda o klasický zápis formou prepisovacích pravidiel. Na ľavej strane pravidla je uvedený neterminál, ktorý musí byť deklarovaný v časti deklarácii. V sekcii `expansion_choices`, sú umiestnené telá pravidiel v BNF notácii a akcie definované programátorom. Telo pravidla sa skladá z deklarovaných symbolov. Príklad pravidla z predchádzajúcej kapitoly, by v CUP-e vyzeral takto:

```
terminal               LEFT, AST, RIGHT;
terminal     String    IDENTIFIER;
non terminal StateList state_list, ident_list;

state_list ::=
  LT AST GT
  {:
    RESULT = new StateList();
    RESULT.setAll();
  :}
  |
  LT ident_list:i GT
  {:
    RESULT = i;
  :}
  ;

ident_list ::=
  IDENTIFIER:t
  {:
    RESULT = new StateList();
    RESULT.addState(t);
  :}
  |
  ident_list:i COMMA IDENTIFIER:t
  {:
    i.addState(t);
    RESULT = i;
  :}
  ;
```

Oproti JavaCC pribudol blok deklarácií symbolov gramatiky. V JavaCC bolo potrebné na začiatku bloku pravidiel gramatiky uviesť návratový typ a názov metódy zastupujúcej neterminál. V CUP-e je táto konštrukcia nahradená práve deklaráciou neterminálov. V JavaCC sú všetky terminály definované ako inštancie triedy `Token`. V CUP-e je možné určiť typ hodnoty terminálu. Programátorovi nie je v akcii predaný odkaz na inštanciu tokena, ale len na hodnotu tokena. Hodnota tokena je určená pri generovaní tokena v lexikálnom analyzátore. V príklade lexikálneho pravidla, je hodnotou tokena jeho reťazec. Na príklade syntaktického pravidla je takisto vidieť prepis zápisu pravidla v EBNF notácii do zápisu s využitím ľavej rekurzie (pravidlo `ident_list`). V každej akcii je automaticky deklarovaná premenná `RESULT`, do ktorej je v akcii priradená hodnota neterminálu na ľavej strane pravidla.

6.2 Analýza vygenerovaného parsera

Výstupom JFlex-u je zdrojový kód kompatibilný s Javou 1.5. Výstupom CUP-u je zdrojový kód kompatibilný s Javou 1.4. V tejto časti bude popísaná architektúra vygenerovaného parsera. Na nasledujúcom obrázku je znázornený diagram vygenerovaných tried.



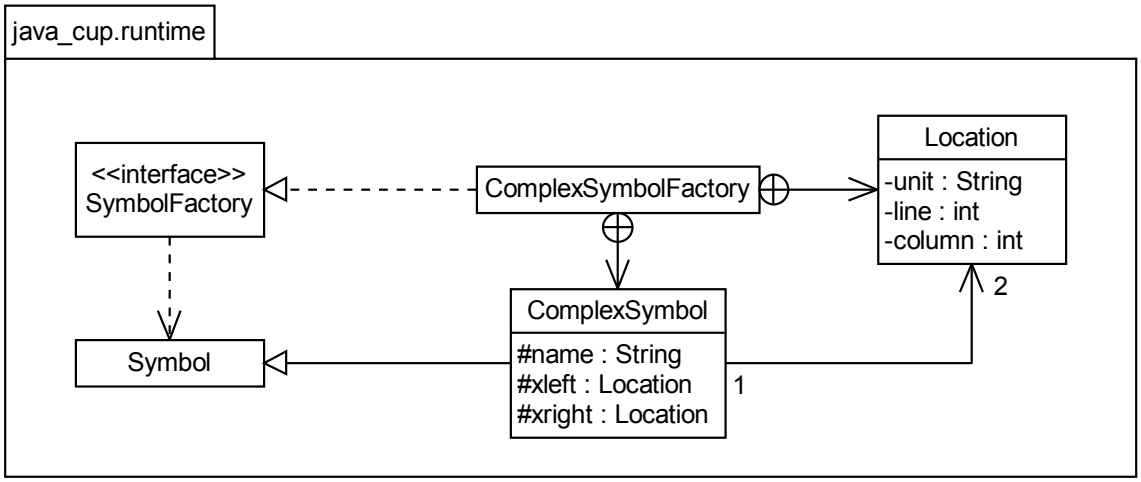
Obrázok 2: Diagram tried vygenerovaného parsera.

6.2.1 Trieda Symbol

Rozhranie `SymbolFactory` a trieda `Symbol` sú umiestnené v balíku `java_cup.runtime`, ktorý je súčasťou CUP-u a je nutné ho pripojiť k vygenerovanému parseru. Inštancie triedy `Symbol` sú použité na vytváranie tokenov

v lexikálnom analyzátore, ale takisto na vytváranie neterminálov v syntaktickom analyzátore. Atribút `sym` obsahuje identifikátor symbolu. Atribúty `parse_state` a `used_by_parser` sú využívané syntaktickým analyzátorom. Prvý uchováva stav analyzátora, druhý slúži ako príznak, či nedošlo k znovupoužitiu inštancie programátorom. Atribúty `left` a `right`, pôvodne slúžili na uchovanie začiatočnej a koncovej pozície symbolu, ale v súčasnosti sú už nahradené samostatnou triedou `Location`, takže existujú už len kvôli spätnej kompatibilite. Atribút `value`, reprezentuje hodnotu symbolu. Hodnota je uchovávaná ako inštancia generického typu `Object`. Pri získavaní hodnoty symbolu, dôjde k pretypovaniu na skutočný typ hodnoty. Toto riešenie neumožňuje uchovávanie primitívnych typov, ale nie je nutné mať pre každý typ samostatnú triedu.

Rozhranie `SymbolFactory` poskytuje metódu `newSymbol` na vytváranie inštancií symbolov. Použitie rozhrania umožňuje skryť implementáciu uchovávaných pozícií symbolu. V CUP-e je totiž možné zapnúť propagovanie pozícií, čo znamená, že v syntaktickom analyzátore budú mať určenú pozíciu aj neterminály. Tá je odvodená od začiatočnej pozície prvého symbolu pravidla a koncovej pozície posledného symbolu pravidla. Pozície sú užitočné v prípade vytvárania abstraktného syntaktického stromu, keď je možné určiť pozíciu celého podstromu, definovaného neterminálnom. Syntaktický analyzátor volá metódu `newSymbol` s parametrami prvého a posledného symbolu pravidla. Na nasledujúcom obrázku je zobrazená trieda implementujúca rozhranie `SymbolFactory`, vrátane rozšírenej triedy `Symbol`. Tieto triedy sú súčasťou balíku `java_cup.runtime`.



Obrázok 3: Implementácia rozhrania SymbolFactory.

Inštancia triedy `ComplexSymbolFactory` je predaná v konštruktoroch lexikálneho a syntaktického analyzátora. Pomocou tejto inštancie, sa volaním metódy `newSymbol` vytvárajú nové symboly. Trieda `ComplexSymbol` rozširuje triedu `Symbol` o atribúty `xleft` a `xright`, ktoré reprezentujú pozíciu symbolu a atribút `name`, ktorý reprezentuje reťazcový identifikátor symbolu, ktorý sa používa v hláseniach o výskyte chýb vo vstupných dátach. Pozície sú imeplementované pomocou inštancií triedy `Location`. Metóda `newSymbol`, implementovaná v triede `ComplexSymbolFactory`, vracia inštancie triedy `ComplexSymbol`.

6.2.2 Lexikálny analyzátor

Trieda `JFlexAnalyzer`, vygenerovaná z JFlex-u, implementuje lexikálny analyzátor. Komunikácia medzi lexikálnym a syntaktickým analyzátorom prebieha prostredníctvom rozhrania `Scanner`, ktoré trieda lexikálneho analyzátora implementuje. Lexikálny analyzátor poskytuje v lexikálnej akcii premenné a metódy, ktoré reprezentujú informácie o rozpoznanom lexéme. Volaním metódy `yytext`, dôjde k vytvoreniu inštancie triedy `String` so znakovými dátami rozpoznaného lexému. Premenné `yyline` a `yycolumn`, obsahujú čísla riadku a stĺpca, na ktorom rozpoznaný lexém začína. Oproti JavaCC sa teda jedná o odlišný prístup. Namiesto automatického vytvorenia inštancie tokena, vrátane jeho reťazca, je programátorovi poskytnutá sada metód, prípadne premenných, pomocou ktorých je možné token vytvoriť.

Kód starajúci sa o správu vstupných dát, je súčasťou triedy lexikálneho analyzátora. Princípy popísané v JavaCC pre triedu `CharStream` sú rovnaké aj v JFlex-e. Oproti JavaCC však nie je použité pole obsahujúce čísla riadkov a stĺpcov. Po rozpoznaní lexému a vykonaní akcie, dôjde k spracovaniu rozpoznaného reťazca, pri ktorom je dopočítané číslo riadku a stĺpca, na ktorom bude začínať nasledujúci rozpoznaný reťazec. Vracanie znakov späť do buffera buď v lexikálnom analyzátore alebo v akcii teda nevadí, ale v akcii nie je poskytnutá koncová pozícia reťazca. Namiesto použitia cyklického buffera, sa znaková sekvencia práve rozpoznávaného reťazca skopíruje na začiatok znakového poľa buffera a nové dáta sa načítajú za túto sekvenciu. Pri tomto prístupe je vždy znaková sekvencia rozpoznaného reťazca súvislá.

JFlex používa implicitne buffer o veľkosti 16 384 znakov. Algoritmus expanzie buffera je oproti JavaCC trochu iný. Namiesto zväčšenia veľkosti buffera o zvolenú konštantu, sa veľkosť buffera zdvojnásobí. To umožňuje rýchlejšie dosiahnutie požadovanej veľkosti, za cenu možného nevyužitia buffera. Pri reinicializácii lexikálneho analyzátora, je používaný buffer zachovaný.

Rozhranie `Sym` je generované z CUP-u, a obsahuje definície identifikátorov tokenov. Tie sú získané z deklarácie terminálov v špecifikácii syntaktického analyzátora.

6.2.3 Syntaktický analyzátor

CUP poskytuje balík `java_cup.runtime`, ktorý obsahuje triedy s všeobecným kódom, použiteľným pre každý vygenerovaný syntaktický analyzátor. Tento balík je potrebné pripojiť k výslednému parseru. V balíku sa okrem tried symbolov a rozhrania pre lexikálny analyzátor, nachádza abstraktná trieda `lr_parser` a trieda `virtual_parse_stack`. Trieda `lr_parser` implementuje základnú logiku parsera. Syntaktický analyzátor si drží aktuálny token v atribúte `cur_token`. Keďže CUP generuje LALR(1) analyzátory, nie je potrebná žiadna ďalšia štruktúra uchovávajúca tokeny. Pri potrebe načítania nového tokena, sa zavolá metóda `next_token` a odkaz na token sa zapíše do atribútu `cur_token`. Syntaktický analyzátor používa explicitný zásobník na uchovávanie symbolov. Tento zásobník je deklarovaný v triede `lr_parser` a je implementovaný pomocou triedy `Stack`, ktorá je súčasťou štandardného balíku `java.util`. Dosiahnuté výsledky testov kolekcií z kapitoly 4 ukazujú, že použitie štandardnej triedy `Stack` na implementáciu zásobníka, so sebou prináša výraznú réžiu. Oproti JavaCC je takisto nutné vytvárať nové inštancie triedy `Symbol` aj pre neterminály.

Trieda `CUPAnalyzer`, ktorá je odvodená od triedy `lr_parser`, je vygenerovaná CUP-om zo špecifikácie syntaktického analyzátora. Trieda obsahuje tabuľky automatu syntaktického analyzátora a akcie.

6.2.4 Zotavenie z chýb

Mechanizmus na zotavenie z chýb je podobný ako v JavaCC. V miestach kde je potrebné reagovať na výskyt chýb, sa definujú pravidlá so špeciálnym symbolom `error`. Pri výskyte chyby, dôjde k odstráneniu symbolov zo zásobníka, až kým sa nenarazí na stav, z ktorého je definovaný prechod pre symbol `error`. Tento symbol sa vloží na zásobník a syntaktický analyzátor pokračuje v simulačnej fáze. Počas tejto fázy je vykonávaná analýza vstupných dát bez spúšťania akcií. Simulačná fáza končí, keď sa podarí spracovať určitý počet tokenov na vstupe. V prípade, že pri spracovávaní tokenov dôjde k chybe, vstup sa posunie o jeden token dopredu a simulačná fáza sa reštartuje. Aby bolo možné tokeny opakovane spracovávať, sú udržované v poli. V simulačnej fáze je takisto potrebný samostatný zásobník, aby zostal obsah pôvodného zásobníka zachovaný. Tento zásobník je implementovaný v triede `virtual_parse_stack`. Po úspešnom ukončení simulačnej fázy, dôjde k spracovaniu tokenov ešte raz, ale aj s vykonaním akcií a nad normálnym zásobníkom. Potom sa syntaktický analyzátor vráti k normálnej analýze. Príklad z JavaCC, by v CUP-e vyzeral takto:

```
stm ::= while_stm SEMI | if_stm SEMI | error SEMI;
```

Definovanie symbolu `error` a rušenie obsahu zásobníka pri vzniku chyby, je možne prirovnať k mechanizmu výnimiek v JavaCC. Oproti JavaCC však prebieha zotavenie z chyby automaticky.

7 Návrh

V predchádzajúcej kapitole bol predstavený generátor parserov JFlex/CUP, ktorý je ekvivalentný produktom Flex/Bison pre jazyk C. Obidva projekty sú vyvíjané pomerne dlhú dobu. Licencie, pod ktorými sú distribuované, umožňujú vytváranie a zverejňovanie modifikovaných verzií. JFlex naviac umožňuje vytvárať analyzátory s plnou podporou Unicode. Medzi ďalšie príjemné vlastnosti, ktoré sa v nástroji Flex nenachádzajú, patrí napríklad automatický výpočet pozície rozpoznávaných reťazcov alebo nové konštrukcie v regulárnych výrazoch, ktoré uľahčujú ich zápis. CUP oproti Bisonovi neumožňuje vytváranie GLR *(Generalized LR)* syntaktických analyzátorov. GLR syntaktický analyzátor umožňuje spracovanie ľubovoľnej bezkontextovej gramatiky. Keďže na väčšinu bežne spracovávaných jazykov stačí LALR(1) analyzátor, je využite tohto prostriedku okrajové. Ostatné vlastnosti sú s Bisonom porovnateľné. Na základe týchto poznatkov je zrejmé, že vytvárať od základov nový generátor parserov, by bolo neefektívne. Napríklad implementácie generujúce konečné automaty, buď z regulárnych výrazov v lexikálnom analyzátore alebo z pravidiel gramatiky v syntaktickom analyzátore, sú hotové a ich opätovná reimplementácia by nepriniesla nič nové.

Nová implementácia generátora parserov bola preto založená na JFlexe-e a CUP-e. Nasleduje popis častí, ktorých sa mala modifikácia dotknúť.

7.1 Rozhranie lexikálneho analyzátora voči programátorovi

Najdôležitejšou informáciou, ktorú má lexikálny analyzátor poskytovať programátorovi, je reťazec, ktorý bol v zhode nájdený. Programátor môže s týmto reťazcom naložiť troma rôznymi spôsobmi:

Ignorovanie reťazca

V takýchto prípadoch je zbytočné vytvárať novú inštanciu triedy `String`, tak ako to je implementované v JavaCC. JFlex umožňuje vytvoriť reťazec na požiadanie, volaním metódy `yytext`.

Využitie reťazca v lexikálnej akcii

Typicky sa jedná o transformáciu reťazca do inej podoby. Napríklad to môže byť zápis reťazca do inštancie triedy `StringBuffer`, kvôli modifikáciám, prípadne spájaniu viacerých reťazcov, získanie identifikátora reťazca z hašovacej tabuľky, alebo konverzia reťazca na číslo. JavaCC, ale aj JFlex, vyžadujú v týchto prípadoch vytvorenie novej inštancie triedy `String`, aby sa bolo možné dostať k znakovým dátam rozpoznaného reťazca. Inštancia triedy `String`, slúži ako prechodná reprezentácia nájdeného reťazca. V prvom prípade sa zmení na znakové dáta v inštancii triedy `StringBuffer`, v druhom na číselný identifikátor reťazca a v treťom na číslo.

V takýchto prípadoch, by bolo možno výhodnejšie dovoliť prístup k znakovým dátam nájdeného reťazca vo vstupnom buffery. Programátor by mal v lexikálnej akcii k dispozícii odkaz na vstupný buffer, vrátane počiatočného a koncového indexu znakovej sekvencie v buffery, ktorá tvorí nájdený reťazec.

Pri predpoklade, že je vstupný buffer implementovaný ako znakové pole, teda je to inštancia triedy `char[]`, by sme v prvom príklade ušetrili vytváranie novej inštancie triedy `String`. Trieda `StringBuffer` poskytuje aj rozhranie pre prácu so znakovými poľami. Za predpokladu, že je hašovacia tabuľka implementovaná ako inštancia triedy `Hashtable`, by sme v druhom príklade narazili na problém. Trieda `Hashtable` umožňuje uchovávať iba inštancie objektových typov. Informácia predaná programátorovi, je zložená z troch atribútov. Pre takéto situácie, je teda potrebné poskytnúť programátorovi možnosť vytvorenia objektu, ktorý obaľuje tieto atribúty. Oproti vytvoreniu triedy `String`, by nebolo nutné vytvárať znakové pole a kopírovať do neho znakové dáta z buffera. Pre tretí príklad, by bolo možné na konverziu reťazca na číslo, použiť metódu `valueOf` triedy `Integer`. Táto metóda však vyžaduje reprezentáciu reťazca v inštancii triedy `String`. Pre takéto situácie, by bolo vhodné poskytnúť vlastnú implementáciu bežne používaných metód pracujúcich s reťazcami. Tieto metódy by umožňovali prácu s reprezentáciou reťazca v znakovom poli.

Predanie reťazca syntaktickému analyzátoru

Takisto ako v predchádzajúcom prípade, je v JFlex-e a v JavaCC, nutné vytvoriť inštanciu triedy `String`. Navrhnutú reprezentáciu nájdeného reťazca, v tomto prípade nie je možné použiť. Syntaktický analyzátor uchováva tokeny kvôli výhľadu, prípadne na zásobníku pri spracovávaní pravidla. Takže kým sa dostane nájdený reťazec k programátorovi v akcii syntaktického analyzátora, môžu byť od lexikálneho analyzátora získané ďalšie tokeny. Môže sa stať, že v lexikálnom analyzátore dôjde k načítaniu nových dát do buffera a k zrušeniu jeho predchádzajúceho obsahu. V prípade potreby uchovania reťazca po určitú dobu, by mohlo byť výhodné povoliť vytvorenie referencie na vstupný buffer lexikálneho analyzátora. V prípade potreby načítania nových dát do buffera, na ktorý by existovali referencie, by bolo možné postupovať nasledujúcimi spôsobmi.

- zavolanie obslužnej rutiny, ktorá by zabezpečila uvoľnenie referencií na buffer a vytvorenie kópií znakových dát reťazcov. Po vykonaní tejto akcie, by bolo možné obsah buffera prepísať.
- nový buffer by sa získal z poolu bufferov. V prípade, že by sa v poole nenachádzali žiadne buffery, bol by vytvorený nový buffer. Po skončení využívania buffera, uvoľnením referencií na neho, by došlo k vráteniu buffera do poolu. Buffery, ktoré by sa do poolu po vrátení nezmestili, by boli zahodené.
- kombinácia vyššie uvedených postupov. Opäť by bol využitý pool bufferov. Po využití určitého množstva pamäte pre buffery, by pri potrebe nového buffera, v prípade, že by sa v poole nenachádzal žiaden buffer, došlo k zavolaniu obslužnej rutiny ako v prvom prípade, ktorá by zabezpečila uvoľnenie referencií a vrátenie nejakého buffera do poolu. Cieľom tohto prístupu by malo byť zabránenie vytvárania bufferov, ktoré by presiahli kapacitu poolu a pri vracaní by došlo k ich zahadzovaniu.

Použitie uvedených riešení je obmedzené na prípady, keď nie je nutná reprezentácia reťazca v inštancii triedy `String`. V prípade použitia bufferov s vytváraním referencií, je nutné explicitné uvoľňovanie referencií programátorom. Na základe týchto obmedzení bola preto naviac navrhnutá nasledujúca metóda.

7.1.1 Správa inštancií triedy String

Trieda `String` predstavuje štandardnú reprezentáciu reťazca v jazyku Java. Preto veľká časť knižníc umožňuje prácu s reťazcami iba pomocou tejto reprezentácie. Inštancie triedy `String` naviac patria k nemenným objektom. Takže nie je možné zdieľanie znakového poľa reťazca so vstupným bufferom lexikálneho analyzátora. Oproti vytváraniu kópií znakových dát z buffera pre každý nájdený reťazec, by bolo možné vytvoriť nad spracovávaným úsekom buffera jednu inštanciu triedy `String`, ktorá by slúžila ako keš. Pre nájdené reťazce by sa inštancie vytvárali volaním metódy `substring` na keš. Pri volaní metódy `substring`, nedochádza k vytváraniu kópie znakových dát, ale tie sú zdieľané s pôvodnou inštanciou. Oproti pôvodnej variante by tak bolo možné sústrediť znakové dáta a ich inicializáciu na jednom mieste.

Využitie uvedenej metódy však záleží na implementácii metódy `substring`. Nikde totiž nie je zaručené, že popísaná implementácia bude platiť aj v nasledujúcich verziách Javy. Keďže keš je implementovaná ako inštancia triedy `String`, je po prepísaní obsahu buffera nutné vytvoriť novú inštanciu a pôvodnú zahodiť.

7.2 Rozhranie medzi lexikálnym a syntaktickým analyzátorom

Komunikácia medzi lexikálnym a syntaktickým analyzátorom prebieha prostredníctvom tokenov. V JavaCC a v CUP-e je token implementovaný ako inštancia triedy, ktorá uchováva identifikátor a hodnotu tokena, prípadne ďalšie informácie. Pri vygenerovaní tokena je nutné vytvoriť vždy novú inštanciu. V CUP-e je pri typickom použití programátorovi v akcii sprístupnená hodnota tokena a pozícia vo vstupných dátach.

Na token je možné sa pozrieť ako na prechodnú reprezentáciu týchto informácií. Namiesto vytvárania inštancie pre každý vygenerovaný token, by syntaktický analyzátor mohol poskytovať rozhranie, pomocou ktorého by programátor predal v lexikálnej akcii syntaktickému analyzátoru informácie o zhode. Syntaktický analyzátor by si interne udržiaval množinu objektov reprezentujúcich tokeny, ktoré by sa opakovane používali. V CUP-e sa vytvárajú nové inštancie aj pre neterminálne symboly. Množina objektov využívaných na uchovávanie tokenov, by mohla byť použitá aj pre neterminály.

7.3 Implementácia zásobníka syntaktického analyzátora

Ako už bolo uvedené v predchádzajúcej kapitole, je v CUP-e zásobník vygenerovaného syntaktického analyzátora implementovaný pomocou triedy `Stack`, balíka `java.util`. Na základe výsledkov testov zo 4. kapitoly, sa ukazuje ako výhodná vlastná implementácia zásobníka pomocou poľa špecifického typu. Okrem zníženia celkovej réžie pri použití štandardnej triedy `Stack`, tak bude možné implementovať zásobník `virtual_parse_stack` pomocou poľa uchovávajúceho položky typu `int`.

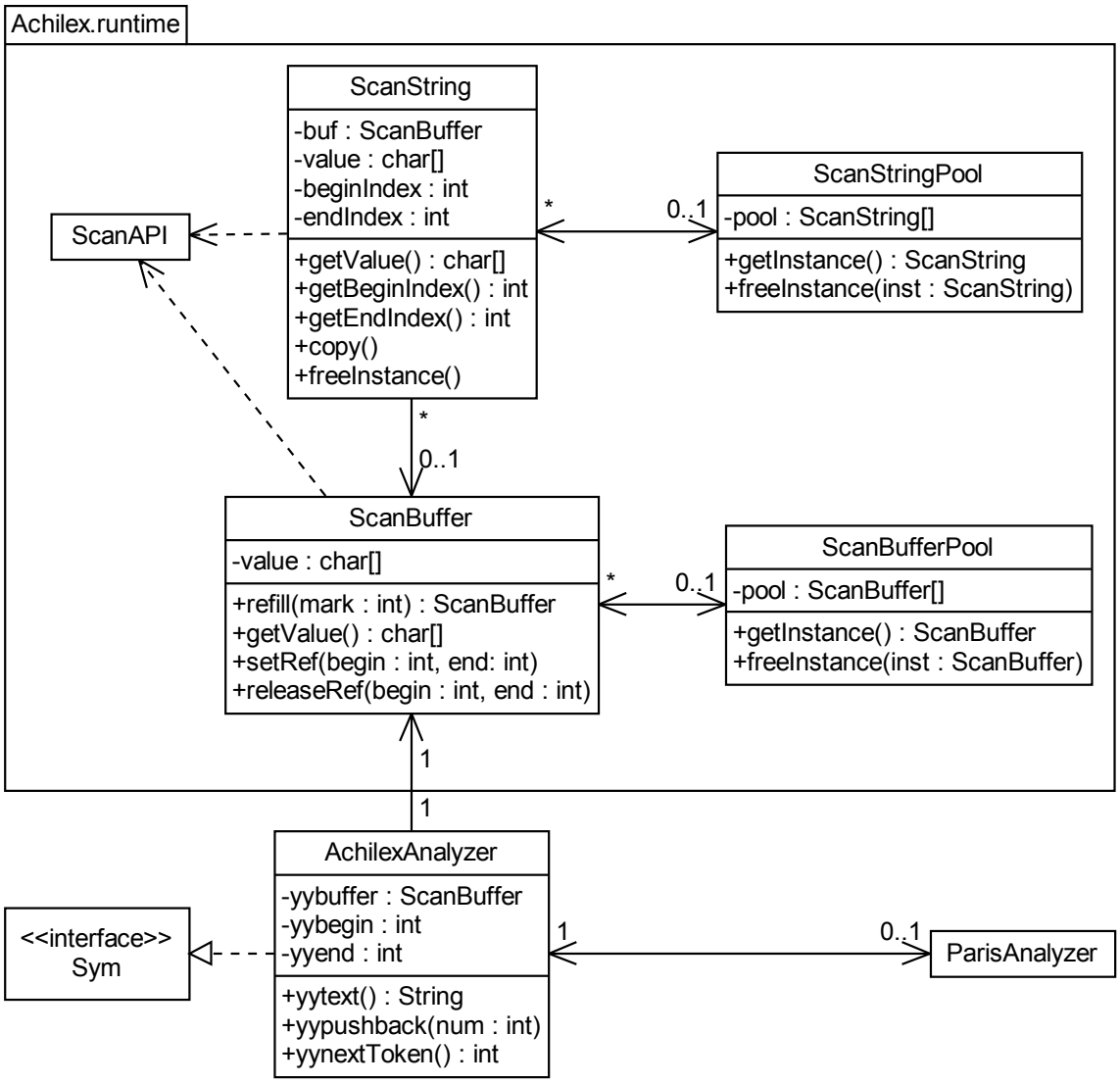
8 Achilex/Paris

Podľa návrhu uvedeného v predchádzajúcej kapitole, bol implementovaný generátor lexikálnych analyzátorov pod názvom Achilex, založený na JFlex-e vo verzii 1.4.1 a generátor syntaktických analyzátorov pod názvom Paris, založený na CUP-e vo verzii 11a beta 20060330. V čase písania diplomovej práce, bola vytvorená pre Achilex verzia 1.1.3 a pre Paris verzia 1.3.1.

8.1 Špecifikácia parsera

Syntax špecifikácie pre lexikálny a syntaktický analyzátor bola prevzatá z JFlex-u a CUP-u. Boli len pridané niektoré parametre, ktoré je možné uviesť v špecifikáciách. Podrobný popis týchto zmien je možné nájsť v priložených dokumentáciách. V nasledujúcich častiach bude popísaná architektúra vygenerovaného parsera. V rámci popisu architektúry, budú popísané aj konštrukcie na manipuláciu s nájdeným reťazcom, tokenom atď.

8.2 Lexikálny analyzátor



Obrázok 4: Diagram tried lexikálneho analyzátora.

Výstupom Achilex-u je zdrojový kód kompatibilný s Javou 1.5. Na obrázku 4 je znázornený diagram tried vygenerovaného lexikálneho analyzátora. Okrem triedy `AchilexAnalyzer`, v ktorej sa nachádza implementácia lexikálneho analyzátora, je súčasťou vygenerovaných zdrojových kódov balík `Achilex.runtime`. Táto knižnica je používaná jednak samotným lexikálnym analyzátorom, ale takisto poskytuje triedy, ktoré sú k dispozícii programátorovi a umožňujú prácu s rozpoznaným reťazcom. Balík je generovaný priamo z programu Achilex. Je možné určiť mnoho parametrov, ktoré ovplyvňujú výslednú podobu balíku. Niektoré parametre budú popísané v texte, ostatné je možné nájsť v priloženej dokumentácii.

Pri pohľade na atribúty a metódy triedy `AchilexAnalyzer` je vidieť, že okrem pôvodného prístupu k rozpoznanému reťazcu pomocou metódy `yytext`, ktorá vytvára inštanciu triedy `String`, poskytuje trieda `AchilexAnalyzer` atribúty `yybuffer`, `yybegin` a `yyend`, pomocou ktorých je umožnený priamy prístup k znakovým dátam reťazca v buffery. Atribút `yybuffer` obsahuje odkaz na inštanciu vstupného buffera. Atribút `yybegin` obsahuje index prvého znaku rozpoznaného reťazca v znakovom poli buffera. Atribút `yyend` obsahuje index nasledujúceho znaku za posledným znakom rozpoznaného reťazca. Vstupný buffer je na rozdiel od JFlex-u implementovaný v samostatnej triede `ScanBuffer`, ktorá je súčasťou balíku `Achilex.runtime`.

V nasledujúcej časti budú popísane jednotlivé triedy balíku `Achilex.runtime`. V záverečnej časti popisu bude uvedený príklad použitia.

8.2.1 Trieda ScanAPI

Táto trieda poskytuje sadu statických metód na prácu s reťazcami. Metódy používajú rozhranie, ktoré umožňuje vykonávanie operácií nad základnou reprezentáciou reťazcov vo forme znakových sekvencií v znakových poliach. Teda každá metóda prijíma reťazec formou trojice atribútov `value`, `beginIndex` a `endIndex`, kde `value` je znakové pole obsahujúce dáta reťazca, `beginIndex` je index prvého znaku reťazca v poli a `endIndex` je index nasledujúceho znaku za posledným znakom reťazca. Väčšina operácií vznikla reimplementáciou metód z triedy `String`. Implementované boli nasledujúce kategórie operácií:

- **operácie nad dvomi reťazcami -** jedná sa o porovnávacie a vyhľadávacie operácie. Implementované boli tieto operácie: `contentEquals`, `compareTo`, `startsWith`, `endsWith`, `indexOf` a `lastIndexOf`. Rozhranie metód umožňuje reprezentáciu druhého reťazca buď pomocou znakového poľa alebo pomocou rozhrania `CharSequence`, čím je napríklad umožnená práca s inštanciami triedy `String`. Pre každú metódu existuje varianta, ktorá umožňuje vykonanie operácie s ignorovaním veľkosti písmen.
- **vyhľadávanie znakov -** implementované boli metódy `indexOf`, `lastIndexOf`, `trimIndex` a `lastTrimIndex`.
- **transformačné operácie -** implementované boli metódy na konverziu veľkosti písmen (`toLower` a `toUpper`) a metódy na konverziu medzi reťazcami a celými číslami (`toInt` a `valueOf`). Transformačné metódy, kde je výsledkom transformácie reťazec, umožňujú buď automatické vytvorenie znakového poľa výsledného reťazca alebo je znakové pole, do ktorého sa znakové dáta skopírujú, jedným z parametrov metódy.

Metódy na prácu s reťazcami sú k dispozícii aj v triedach `ScanBuffer` a `ScanString`. Implementácie v týchto triedach obaľujú volania odpovedajúcich metód v triede `ScanAPI` a poskytujú rozhranie, ktoré umožňuje reprezentáciu reťazcov podľa príslušnej triedy. Programátor má podľa vybranej implementácie reťazcov, tri možnosti ako s nimi pracovať.

- v prípade použitia reprezentácie pomocou znakového poľa, je umožnené priame volanie metód z triedy `ScanAPI`.
- v prípade použitia reprezentácie pomocou inštancie buffera a indexov vymedzujúcich reťazec v buffery, je možné použiť metódy poskytované triedou `ScanBuffer`.
- poslednou možnosťou je využitie reprezentácie pomocou triedy `ScanString`.

Pri generovaní balíku je možné zvoliť variantu, v ktorej nedôjde k vygenerovaniu triedy `ScanAPI`. V prípade, že operácie s reťazcami nie sú využívané, to môže zmenšiť veľkosť vygenerovaných tried.

8.2.2 Trieda ScanBuffer

V tejto triede je implementovaný vstupný buffer lexikálneho analyzátora. Lexikálnemu analyzátoru je umožnený priamy prístup k znakovým dátam buffera. Pri potrebe načítania nových dát, volá lexikálny analyzátor metódu `refill`. Parameter `mark` určuje pozíciu v znakovom poli, od ktorej majú byť znakové dáta zachované. Tieto znakové dáta, sa podobne ako v JFlex-e, skopírujú na začiatok znakového poľa. V prípade, že pri volaní metódy `refill`, došlo k zmene aktívneho buffera na iný, tento nový buffer bude vrátený v návratovej hodnote metódy. Pomocou metódy `getValue`, je možné získať prístup k znakovému poľu buffera a v lexikálnej akcii tak pracovať priamo so znakovými dátami rozpoznaného reťazca. Na základe nastavenia parametrov pri generovaní balíku, je možne vytvoriť tri rôzne implementácie tejto triedy.

Základná implementácia s jedným bufferom

Implementácia buffera je rovnaká ako v JFlex-e. Veľkosť buffera je implicitne nastavená na 8 192 znakov. Oproti JFlex-u je však v lexikálnej akcii umožnený prístup k rozpoznanému reťazcu, bez nutnosti vytvárania inštancie triedy `String`. Je možné využiť trojicu atribútov `yybuffer`, `yybegin` a `yyend` v kombinácii s pokytovanými metódami na prácu s reťazcami, prípadne je možné použiť triedu `ScanString`, ktorá bude popísaná ďalej. Pri potrebe práce s reťazcom mimo lexikálnu akciu, je nutné vytvoriť inštanciu triedy `String`, volaním metódy `yytext`. V tejto variante nie je vygenerovaná trieda `ScanBufferPool` a metódy `setRef` a `releaseRef`.

Implementácia s jedným bufferom s možnosťou vytvárania referencií

Pri použití tejto varianty, je možné zdieľať znakové dáta rozpoznaného reťazca aj mimo lexikálnej akcie. Je to umožnené použitím referencií na buffer. K dispozícii sú metódy `setRef` a `releaseRef`, kde zavolaním metódy `setRef`, dôjde k vytvoreniu referencie a volaním metódy `releaseRef`, zase k uvoľneniu referencie na buffer. Buffer si drží interný čítač, ktorý je na začiatku inicializovaný na hodnotu 0. Po zavolaní metódy `setRef`, sa hodnota čítača zvýši o 1, po zavolaní metódy `releaseRef` dôjde

k jej zníženiu o 1. Pri potrebe načítania nových dát do buffera volaním metódy `refill`, dôjde ku kontrole hodnoty čítača. V prípade, že je hodnota rovná 0, na buffer neexistujú žiadne referencie a jeho obsah je možné prepísať. V opačnom prípade je nutné zabezpečiť uvoľnenie referencií na buffer.

V buffery nie je implementovaná žiadna správa referencií. Uvoľnenie referencií je preto implementované zavolaním obslužnej rutiny, ktorú musí naprogramovať programátor. Keďže referencie môžu byť držané aj na zásobníku syntaktického analyzátora, je zavolaná špeciálna metóda syntaktického analyzátora, ktorá zabezpečí uvoľnenie referencií z príslušných štruktúr. V prípade, že po vykonaní obslužných rutín sa hodnota čítača zníži na 0, je možné obsah buffera prepísať. Inak je nutné vytvoriť novú inštanciu buffera.

Táto varianta sa hodí v prípade krátkodobého využívania reťazcov. Trieda `ScanBufferPool` v tomto prípade nie je k dispozícii.

Implementácia s viacerými buffermi s možnosťou vytvárania referencií

Oproti predchádzajúcej variante, je inak riešená situácia, keď sa pri volaní metódy `refill` zistí nenulový počet referencií na buffer. Namiesto uvoľňovania referencií na buffer, je využitý pool bufferov. Ten je implementovaný v triede `ScanBufferPool`. Na pool je zavolaná metóda `getInstance`, ktorá získa nový buffer z poolu. Do tohto buffera sú potom v metóde `refill` načítané nové dáta zo vstupu a buffer je vrátený lexikálnemu analyzátoru. Takisto je v tejto variante inak implementovaná metóda `releaseRef`. V prípade, že po zavolaní metódy klesne počet referencií na buffer na hodnotu 0 a buffer nie je aktívny (nie je to buffer, ktorý je práve spracovávaný lexikálnym analyzátorom), je na pool zavolaná metóda `freeInstance`, ktorá zabezpečí vrátenie buffera späť do poolu. Pool je implementovaný ako pole, na ktorého koniec sa pridávajú, prípadne odoberajú inštancie. Takáto jednoduchá implementácia zabezpečuje minimálnu réžiu spojenú so správou bufferov.

V špecifikácii lexikálneho analyzátora je možné nastaviť maximálnu kapacitu poolu, v počte znakov udržiavaných bufferov. Implicitne je táto hodnota nastavená na 262 144 znakov. V prípade, že po zavolaní metódy `getInstance`, sa v poole nenachádza žiaden buffer, je nutné vytvoriť novú inštanciu buffera.

Je možné určiť počet znakov, po ktorých naalokovaní pre buffery, v prípade, že po zavolaní metódy `getInstance` sa v poole nenachádza žiaden buffer, dôjde k zavolaniu obslužných rutín na zrušenie referencií. V prípade, že po vykonaní tejto akcie, nedôjde k uvoľneniu nejakého buffera, je nutné vytvoriť novú inštanciu. Implicitne je počet maximálne naalokovaných znakov nastavený rovnako ako kapacita poolu, na 262 144 znakov. Teda v prípade, že dôjde k naplneniu kapacity poolu, sa namiesto alokácie ďalších bufferov spustí mechanizmus, ktorý zabezpečí uvoľnenie bufferov do poolu. Pri nastavení maximálneho počtu naalokovaných znakov na hodnotu menšiu ako je kapacita poolu, je možné využiť rezervné miesto na prípadné expanzie bufferov. Implementácia rušenia referencí je rovnaká ako pri použití jedného buffera.

V prípade, že nie sú definované obslužné rutiny a naalokované buffery presiahnu kapacitu poolu, dôjde pri vracaní bufferov, ktoré sa do poolu nezmestia k ich zahadzovaniu.

Réžia spojená so správou bufferov je minimálna. Samozrejme je problém v prípade, že dôjde k zaplneniu kapacity poolu, kedy je nutné volať odpovedajúce obslužné rutiny, prípadne dochádza k zahadzovaniu bufferov. Preto je dôležité správne nastavenie tejto kapacity, čo môže byť niekedy zložité. Ďalší problém nastáva pri nesprávnom nastavení veľkosti bufferov. Oproti variante s jedným bufferom, keď expandoval len tento jeden buffer, tu môže dôjsť k expanzii viacerých bufferov.

8.2.3 Trieda ScanString

Hlavným účelom tejto triedy je zabalenie reprezentácie reťazca vo forme trojice atribútov: `yybuffer`, `yybegin` a `yyend` do objektovej podoby. Atribút `value` uchováva znakové pole reťazca. Znakové pole reťazca nemusí byť vždy zdieľané s bufferom, preto je dobré mať odkazy na znakové pole a na buffer v oddelených atribútoch. Metódy `getValue`, `getBeginIndex` a `getEndIndex` umožňujú získať priamy prístup k dátam reprezentovaného reťazca. Trieda poskytuje konštruktory, ktoré umožňujú vytvorenie inštancie buď nad trojicou atribútov `yybuffer`, `yybegin` a `yyend` alebo nad znakovým poľom, buď s vytvorením kópie znakových dát v znakovom poli alebo so zdieľaním znakových dát so znakovým poľom. Treťou možnosťou je použiť v konštruktore inštanciu rozhrania `CharSequence` a vytvárať tak inštancie tejto triedy, napríklad z inštancií triedy `String`. Trieda implementuje metódy `equals` a `hashCode`. Jej inštancie je teda možné použiť v kolekciách ako `Hashtable` alebo `HashSet`.

Inštancie je možné uchovávať v poole, ktorý je implementovaný v triede `ScanStringPool`. Metóda `getInstance`, umožňuje získanie inštancie z poolu. Metóda poskytuje rovnaké rozhranie ako konštruktory triedy. Po zavolaní metódy `freeInstance` na inštanciu, dôjde okrem uvoľnenia inštancie do poolu k zavolaniu metódy buffera `releaseRef` v prípade, že znakové dáta inštancie sú zdieľané s bufferom. Pool nemusí byť súčasťou vygenerovaného balíku. V prípade, že nedôjde k jeho vygenerovaniu, dôjde pri zavolaní metódy `freeInstance` iba k zavolaniu metódy `releaseRef`. Pool je implementovaný rovnako ako pool bufferov. Teda sa jedná o pole inštancií. Oproti poolu bufferov však po vrátení inštancie späť do poolu, dôjde k nastaveniu všetkých objektových odkazov na hodnotu `null`. Teda v prípade, že inštancia uchováva znakové pole, ktoré nie je zdieľané s bufferom, dôjde k jeho uvoľneniu.

Metóda `copy` vytvorí explicitnú kópiu znakových dát reťazca v prípade, že sú zdieľané s bufferom. Táto metóda je používaná pri rušení referencií na buffer.

Vyššie uvedená implementácia, platí v prípade, že je použitá varianta implementácie triedy `ScanBuffer` s možnosťou vytvárania referencií. V opačnom prípade, pri zavolaní metódy `freeInstance`, nedochádza k uvoľňovaniu referencií na buffer, prípadne keď nie je použitý pool inštancií triedy `ScanString`, táto metóda nie je k dispozícii. Aj vo variante buffera bez možnosti vytvárania referencií, je možné vytvárať inštancie triedy `ScanString` so znakovými dátami zdieľanými so vstupným bufferom, ale platnosť týchto inštancií musí byť obmedzená na lexikálnu akciu.

8.2.4 Príklad použitia balíku

V tomto príklade bude predstavená základná varianta s jedným bufferom bez vytvárania referencií. V príklade bude predstavené použitie metódy na prácu s reťazcami

a reprezentácia reťazca pomocou triedy `ScanString` s využitím poolu inštancií triedy `ScanString`. Vygenerovanie balíku s uvedenými požiadavkami by malo nasledujúci tvar:

```
java  -jar Achilex.jar
-genlib
-libdir Achilex/runtime
-libpkg Achilex.runtime
-genstrpool
spec.alex
```

Parameter `genlib` povoľuje generovanie balíku. Parameter `libpkg` určuje adresár, v ktorom bude vygenerovaný balík umiestnený a parameter `libpkg` určuje názov vygenerovaného balíku. Parameter `genstrpool` povoľuje vygenerovanie poolu inštancií triedy `ScanString`.

V akcii lexikálneho analyzátora je potrebné z nájdeného reťazca získať jeho identifikátor v hašovacej tabuľke. Ďalej predpokladajme, že nezáleží na veľkosti písmen a v hašovacej tabuľke sú reťazce uložené s veľkými písmenami. V špecifikácii lexikálneho analyzátora je potrebné doimportovať vygenerovaný balík:

```
import Achilex.runtime;
```

Ďalej je potrebné v špecifikácii deklarovať nasledujúce atribúty vygenerovanej triedy lexikálneho analyzátora:

```
private ScanStringPool strPool = new ScanStringPool(1);
private int elArrSize = 128;
private char[] elArr = new char[elArrSize];
```

Pri vytváraní inštancie poolu (atribút `strPool`), je nutné uviesť jeho kapacitu. Keďže v našom príklade budeme poolovať jedinú inštanciu, je kapacita zvolená ako 1. V parametroch programu je takisto možné zvoliť nastavenie, pri ktorom dôjde k vytvoreniu poolu, ktorého kapacita sa bude zväčšovať podľa počtu poolovaných objektov. Samotná akcia by mala nasledujúci tvar:

```
{LETTER} ({LETTER}|{DIGIT})* {

int len = yylength();
int end;
ScanString rec;
Integer val;

if (len > elArrSize) {
  elArr = new char[len];
  elArrSize = len;
}

end = yybuffer.toUpper(yybegin, yyend, 0, elArr);
```

```
rec = strPool.getInstance(elArr, 0, end);
val = elements.get(rec);

if (val == null) {
  elements.put(new ScanString(elArr, 0, end, true), ++max);
  val = Integer.valueOf(max);
}
rec.freeInstance(); }
```

Metóda `toUpper`, ktorú poskytuje trieda vstupného buffera, skonvertuje reťazec definovaný pozíciami v buffery do znakového poľa `elArr`. Následne je volaním `getInstance` z poolu získaná inštancia triedy `ScanString`. Inštancia zdieľa znakové dáta s poľom `elArr`. Pomocou tejto inštancie sa získa identifikátor z hašovacej tabuľky. V prípade, že sa v hašovacej tabuľke reťazec nenachádza, je vytvorená nová inštancia triedy `ScanString`, ktorá si vytvorí kópiu znakových dát zo znakového poľa `elArr` a inštancia sa vloží do hašovacej tabuľky. Poolovaná inštancia je volaním `freeInstance` vrátená do poolu.

Oproti variante s vytváraním inštancií triedy `String`, by sme ušetrili vytváranie dvoch inštancií triedy `String` v prípade, že sa nájdený reťazec v hašovacej tabuľke nachádza, inak by sme ušetrili vytváranie jednej inštancie.

8.2.5 Správa inštancií triedy String

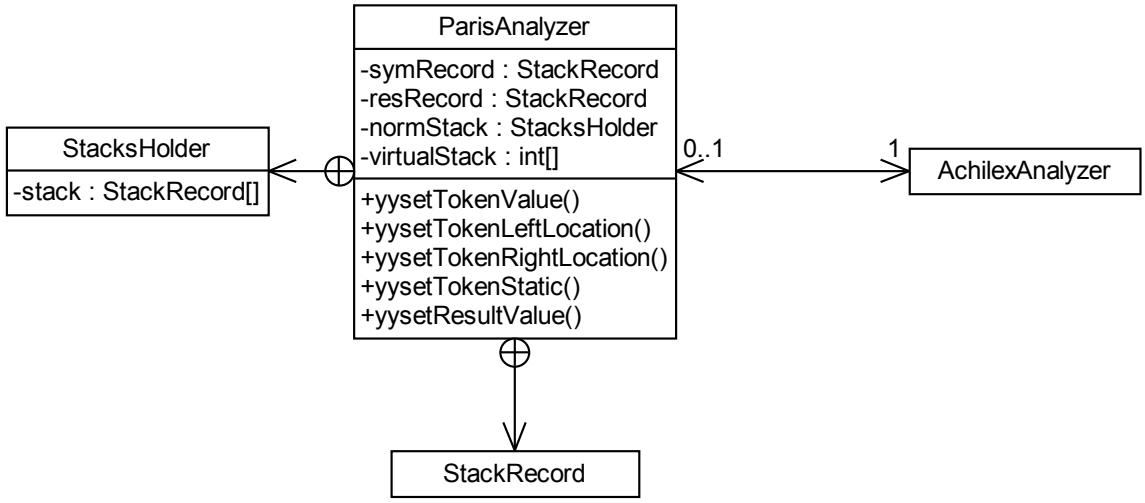
Správu inštancií je možné zapnúť príkazom `%strcache` v špecifikácii lexikálneho analyzátora. Za príkazom je uvedená veľkosť keše. Pri použití tejto voľby je zmenená implementácia metódy `yytext` tak, že nedochádza k vytváraniu inštancie triedy `String` zo znakových dát buffera, ale volaním metódy `substring` na keš. V prípade, že po zavolaní metódy nie je keš k dispozícii, prípadne v keši nie sú aktuálne dáta, je zo znakových dát buffera od pozície `yybegin` vytvorená nová keš o zadanej veľkosti a stará je zahodená. Po zavolaní metódy `refill`, ktorá sa stará o načítanie nových dát do buffera, dôjde zároveň k zahodeniu keše. Implementácia vytvorenia keše spolu s vrátením požadovaného reťazca má v metóde `yytext` nasledujúci tvar:

```
zzstrCache = new String(zzBufferCharArray, begin, cacheSize);
zzCacheStartPos = begin;
zzCacheEndPos = begin + cacheSize;
return zzstrCache.substring(0, actualSize);
```

Atribúty `zzCacheStartPos` a `zzCacheEndPos`, uchovávajú pozície kešovaných znakových dát z buffera, aby tak mohla byť zistená platnosť keše.

8.3 Syntaktický analyzátor

Výstupom Paris-u je zdrojový kód kompatibilný s Javou 1.5.



Obrázok 5: Diagram tried syntaktického analyzátora.

8.3.1 Trieda StackRecord

Na reprezentáciu symbolov (terminálov a neterminálov), používaných v syntaktickom analyzátore slúži trieda `StackRecord`. Jedná sa o reimplementáciu triedy `Symbol` z CUP-u. Oproti CUP-u, však implementácia nie je dopredu daná, ale závisí na požiadavkách uvedených v špecifikácii syntaktického analyzátora. Parametre určujúce implementáciu triedy sú nasledujúce.

Typy hodnôt symbolov

Podobne ako v CUP-e, je možné v príkazoch `terminal` a `non terminal`, uviesť typy hodnôt terminálov a neterminálov. Interne, v triede `Symbol`, je táto hodnota vždy reprezentovaná ako inštancia generického typu `Object` a pri získaní hodnoty symbolu v akcii syntaktického analyzátora, dôjde k pretypovaniu hodnoty na potrebný typ. Toto riešenie neumožňuje použitie primitívnych typov a je potrebné pretypovávanie pri získavaní hodnoty symbolu. Paris umožňuje okrem tohto základného riešenia nasledujúce vylepšenia:

- Namiesto využitia objektových variánt primitívnych typov, je možné priamo využiť primitívne typy pre hodnoty symbolov.
- Je možné eliminovať pretypovávanie pri získavaní hodnôt symbolov, využitím reprezentácie hodnoty v triede `StackRecord`, pomocou špecifického typu. Použitie špecifického typu je možné určiť príkazom `set type`.

Pre typy uvedené ako typ hodnoty pre terminály a neterminály, je implementované nasledujúce mapovanie na typy atribútov v triede `StackRecord`.

1. V prípade, že je použitý jediný objektový typ, je mapovaný sám na seba. Inak sú všetky objektové typy mapované na generický typ `Object`.
2. Primitívne typy `int`, `short`, `byte` a `char`, sa mapujú na najvšeobecnejší typ z tejto množiny, ktorý je použitý pre nejaký terminál alebo neterminál. Teda keď bude použitý typ `int`, všetky typy budú mapované na tento typ.
3. Pre typy `float` a `double`, je použitý rovnaký algoritmus ako v predchádzajúcom bode.
4. Typy `long` a `boolean`, sa mapujú samy na seba.

Pre každý typ, na ktorý existuje mapovanie, sa vytvorí v triede `StackRecord` jeden atribút tohto typu. Základné mapovanie je možné zmeniť použitím príkazu `set type`. Za príkazom je uvedený typ atribútu v triede `StackRecord` a potom nasleduje zoznam typov, ktorých hodnoty sa budú na tento atribút mapovať. Tento príkaz je možné využiť aj na mapovanie rôznych objektových typov na atribúty rôznych typov v triede `StackRecord` a eliminovať tak spomínané pretypovávanie pri získavaní hodnoty. Detailný popis chovania pri zmene implicitného mapovania je možné nájsť v dokumentácii.

Modifikáciou štandardného chovania však rastie počet atribútov v triede `StackHolder` a teda dochádza k vytváraniu väčších objektov. Nevýhodou pri použití mapovania objektových typov na atribúty špecifického typu, je nutnosť nastavovania viacerých atribútov na hodnotu `null`, pri znovu používaní inštancií symbolov.

Definícia reprezentácie pozícií symbolov

V CUP-e je implicitne použitá implementácia pomocou triedy `Location`. Iné implementácie je možné vytvoriť definovaním vlastnej triedy, odvodenej od triedy `Symbol`. V Paris-e je k dispozícii príkaz `locations`, po ktorého použití je možné uviesť zoznam deklarácii premenných. Pre každú premennú budú vytvorené v triede `StackRecord` dva atribúty určeného typu. Jeden pre informáciu o začiatočnej pozícii symbolu, druhý pre informáciu o koncovej pozícii symbolu.

Definícia statických hodnôt

Jedná sa o hodnoty definované pre každý terminál a neterminál. Typickým príkladom je reťazcový identifikátor symbolu, ktorý je použitý pri hlásení o chybách. Pre ich použitie, je k dispozícii príkaz `static`, za ktorým nasleduje zoznam deklarácií premenných. Pre každú premennú bude vytvorený samostatný atribút v triede `StackRecord`.

Ostatné atribúty

Trieda `StackRecord` obsahuje vždy atribút pre číselný identifikátor symbolu a atribút pre stav syntaktického analyzátora.

8.3.2 Príklad implementácie triedy StackRecord

V nasledujúcej časti budeme pracovať s príkladom použitým v kapitole 6, v ktorom bol spracovávaný zoznam stavov lexikálneho analyzátora pre JFlex/CUP. Deklarácia terminálov a neterminálov je doplnená o terminálne symboly `NUMBER` a `CHARACTER`, aby bol viditeľný algoritmus mapovania. Ďalej pribudlo explicitné určenie reťazcového identifikátora a implementácia informácií o pozíciách.

```
terminal                LEFT, AST, RIGHT;
terminal      String    IDENTIFIER;
terminal      int       NUMBER;
terminal      char      CHARACTER;
non terminal StateList state_list, ident_list;
static        String name;
locations     int line, int column;
```

Výsledkom tejto deklarácie bude nasledujúca implementácia triedy `StackRecord`. V komentároch je uvedený popis čo daný atribút z uvedenej deklarácie zastupuje.

```
public class StackRecord {
  public int    val0; //stav
  public int    val1; //identifikátor symbolu
  public int    val2; //locations, left line
  public int    val3; //locations, left column
  public int    val4; //locations, right line
  public int    val5; //locations, right column
  public String val6; //static, name
  public Object val7; //hodnoty typov String a StateList
  public int    val8; //hodnoty typov int a char
}
```

8.3.3 Rozhranie poskytované syntaktickým analyzátorom

Trieda `StackRecord` nie je lexikálnemu analyzátoru dostupná. Namiesto toho je poskytovaná sada metód, pomocou ktorých je možné informácie o tokene predať syntaktickému analyzátoru. Najprv je potrebné lexikálnemu analyzátoru predať inštanciu triedy syntaktického analyzátora. To je možné realizovať použitím parametra v špecifikácii lexikálneho analyzátora, `%useparser`, za ktorý je potrebné uviesť názov triedy syntaktického analyzátora. Inštanciu syntaktického analyzátora je potom možné lexikálnemu analyzátoru predať volaním metódy `yysetParser`. V akcii lexikálneho analyzátora je k dispozícii atribút `yyparser`, ktorý obsahuje odkaz na inštanciu syntaktického analyzátora. Pre príklad z predchádzajúceho odstavca, by sada metód na predanie informácií o tokene vyzerala takto:

```
void yysetTokenStatic(String name);
void yysetTokenLeftLocation(int line, int column);
void yysetTokenRightLocation(int line, int column);
void yysetTokenValueString(String val);
void yysetTokenValueChar(char val);
void yysetTokenValueInt(int val);
```

Metóda `yysetTokenStatic` je vygenerovaná pri použití príkazu `static`. Zoznam parametrov je identický so zoznamom deklarácií uvedeným v tomto príkaze. Metódy `yysetTokenLeftLocation` a `yysetTokenRightLocation` sú vygenerované pri použití príkazu `locations`. Prvá metóda slúži na predanie informácie o začiatočnej pozícii tokena, druhá na predanie informácie o koncovej pozícii tokena. Zoznam parametrov metód je identický so zoznamom deklarácií uvedeným v príkaze `locations`. Nakoniec nasleduje trojica metód na nastavenie hodnoty tokena podľa špecifického typu. Namiesto využitia techniky preťažovania metód, bol zvolený prístup kde typ hodnoty tokena je súčasťou názvu metódy. Je tak zabránené vzniku chýb pri výskyte navzájom pretypovateľných typov. Keďže typ hodnoty tokena môže obsahovať znaky, ktoré nie sú v názvoch metódy povolené, je možné použitím príkazu `method`, určiť názov typu ktorý sa použije pre názov metódy. Tento príkaz je bližšie opísaný v dokumentácii. Príklad lexikálneho pravidla uvedený v kapitole 6, by v tomto prípade vyzeral takto:

```
{LETTER} ({LETTER}|{DIGIT})*
{
  yyparser.yysetTokenStatic("IDENTIFIER");
  yyparser.yysetTokenLeftLocation(yyline + 1,
  yycolumn + 1);
  yyparser.yysetTokenRightLocation(yyline + 1,
  yycolumn + yylength());
  yyparser.yysetTokenValueString(yytext().toUpperCase());
  return IDENTIFIER;
}
```

Pri výskyte chyby v syntaktickom analyzátore, je potrebné podať chybovú správu s informáciami o tokene, na ktorom k chybe došlo. Pre tieto účely k uvedeným set metódam poskytuje syntaktický analyzátor sadu get metód. V prípade, že odpovedajúca set metóda má viac parametrov, je pre každý parameter vygenerovaná jedna get metóda, kde sufixom metódy je názov parametra. Definícia chybovej hlášky by s použitím get metód vyzerala takto:

```
err code {:
  System.err.println("Syntax error (" +
  yygetTokenStatic() + " " +
  yygetTokenLeftLocationLine()+ "/" +
  yygetTokenLeftLocationColumn() + " " +
  yygetTokenRightLocationLine() + "/" +
  yygetTokenRightLocationColumn() + ")");
:}
```

Pre priradenie hodnoty neterminálu na ľavej strane pravidla, poskytuje syntaktický analyzátor sadu set metód, podobne ako pre nastavenie hodnôt tokenov. Príklad syntaktického pravidla zo 6. kapitoly, by v tomto prípade vyzeral takto:

```
state_list ::=
  LT AST GT
  {:
    StateList s = new StateList();
    s.setAll();
    yysetResultValueStateList(s);
  :}
  |
  LT ident_list:i GT
  {:
    yysetResultValueStateList(i);
  :}
  ;


ident_list ::=
  IDENTIFIER:t
  {:
    StateList s = new StateList();
    s.addState(t);
    yysetResultValueStateList(s);
  :}
  |
  ident_list:i COMMA IDENTIFIER:t
  {:
    i.addState(t);
    yysetResultValueStateList(i);
  :}
  ;
```

8.3.4 Implementácia zásobníka

Namiesto použitia štandardnej triedy `Stack`, je využitá vlastná implementácia zásobníka. Ten je implementovaný v triede `StacksHolder`, ako pole inštancií triedy `StackRecord`. V atribúte `normStack` je umiestnený odkaz na inštanciu používaného zásobníka.

Atribút `symRecord` slúži na uchovanie výhľadu. Nad týmto atribútom operujú metódy na nastavovanie a získavanie atribútov tokenov. Napríklad metóda `yysetTokenValueString` z predchádzajúceho odstavca, je implementovaná takto:

```
public void yysetTokenValueString(String val) {
  symRecord.val7 = val;
}
```

Atribút `resRecord` slúži na reprezentáciu vytváraného neterminálu pri redukcii. Nad týmto atribútom pracujú metódy predávanie hodnoty neterminálu na ľavej strane pravidla v akciách. Napríklad metóda `yysetResultValueStateList` z predchádzajúceho odstavca, je implementovaná takto:

```
public void yysetResultValueStateList(StateList val) {
  resRecord.val7 = val;
}
```

Namiesto vytvárania nových inštancií pre každý symbol, je znovu využívaná množina inštancií triedy `StackRecord`. Používané inštancie sú uchovávané na zásobníku a v atribútoch `symRecord` a `resRecord`. Životný cyklus týchto inštancií je nasledujúci:

1. V konštruktore syntaktického analyzátora sú vytvorené inštancie pre atribúty `symRecord` a `resRecord`.
2. Po vrátení volania z lexikálneho analyzátora, sú dáta aktuálneho tokena umiestnené v atribúte `symRecord`.
    a. V prípade, že následne dôjde k vykonaniu akcie posunutia, je odkaz na symbol, umiestnený v atribúte `symRecord`, skopírovaný na vrchol zásobníka. V prípade, že je na tejto pozícii v zásobníku uchovávaný odkaz na nevyužívanú inštanciu, je odkaz skopírovaný do atribútu `symRecord`, inak je vytvorená nová inštancia a odkaz na ňu sa skopíruje do atribútu `symRecord`.
    b. Pri akcii redukcie, sú po vykonaní akcie definovanej programátorom, dáta neterminálu umiestnené v atribúte `resRecord`. Dôjde k odstráneniu určitého počtu symbolov z vrcholu zásobníka. Pri tomto odstraňovaní sú inštancie symbolov odkazované zo zásobníka zachovávané, ale pre každú inštanciu dôjde k nastaveniu všetkých jej atribútov odkazujúcich na objekty, na hodnotu `null`. Potom je odkaz na symbol umiestnený v atribúte `resRecord` skopírovaný na vrchol zásobníka. Opäť ako v predchádzajúcom prípade, je odkaz umiestnený v atribúte `resRecord` nahradený odkazom na vrchole zásobníka, prípadne je vytvorená nová inštancia.

Dátová štruktúra zásobníka teda slúži aj ako úložište nepoužívaných inštancií symbolov.

8.3.5 Zotavenie z chýb

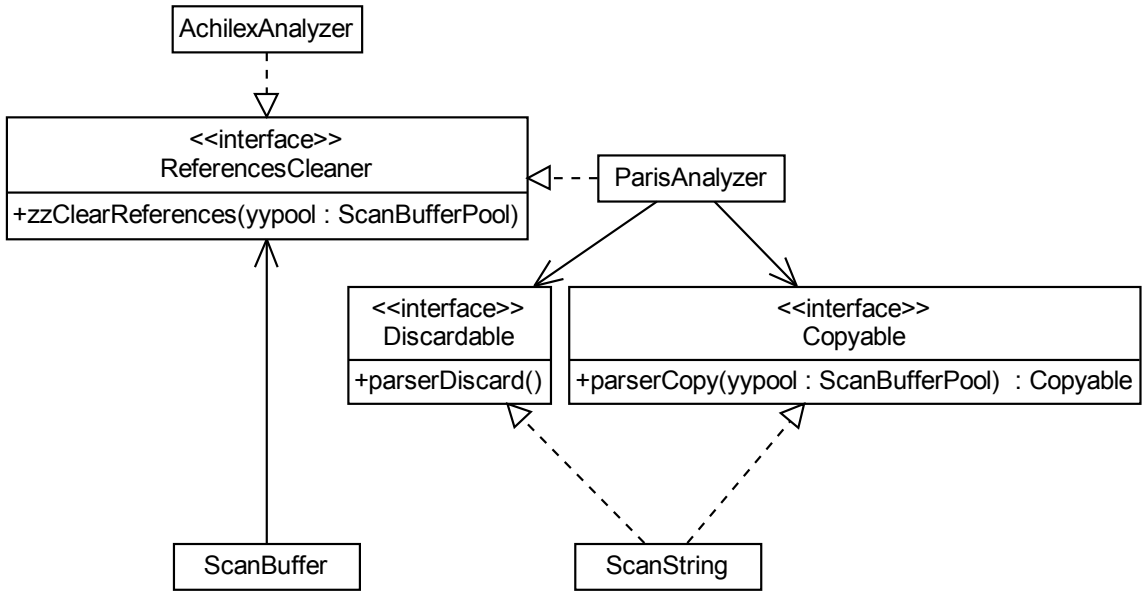
Algoritmus zotavenia z chýb je prebraný z CUP-u. Zásobník používaný v simulačnej fáze však nie je implementovaný ako samostatná trieda s využitím kolekcie `Stack`, ale ako pole typu `int`.

8.4 Generované rozhrania

Keďže Paris nepoužíva žiadnu behovú knižnicu ako CUP, medzi lexikálnym a syntaktickým analyzátorom nemusí byť žiadne rozhranie a je možná priama väzba cez názvy tried.

Pri použití buffera v lexikálnom analyzátore s možnosťou vytvárania referencií, je možné definovať obslužné rutiny, ktoré sa postarajú, v prípade potreby, o zrušenie referencií. Obslužná rutina, ktorá zabezpečí uvoľnenie referencií v interných dátových štruktúrach syntaktického analyzátora, je umiestnená v triede syntaktického analyzátora. Táto obslužná rutina je vygenerovaná automaticky. Obslužná rutina, ktorú definuje programátor, je umiestnená v triede lexikálneho analyzátora. Keďže vygenerovaný balík

`Achilex.runtime` je navrhnutý tak, aby umožňoval využite vo viacerých aplikáciách, je možné spolu s balíkom vygenerovať rozhranie `ReferencesCleaner`, ktoré je potom implementované v triede lexikálneho a syntaktického analyzátora. V rozhraní je deklarovaná metóda `zzClearReferences`, ktorú volá inštancia triedy buffer pri rušení referencií. Pomocou parametra `yypool` je možné zistiť v kóde obslužnej rutiny stav poola bufferov a ukončiť tak spracovávanie rutiny hneď ako dôjde k uvoľneniu aspoň jedného buffera. Pri použití jediného buffera, je v tomto parametri uložená inštancia triedy buffera a v podstate je v tomto prípade umožnené zistiť, či už došlo k uvoľneniu všetkých referencií na buffer. Namiesto triedy `ScanBufferPool` je takisto možné použiť rozhranie a tým odstrániť závislosť rozhrania `Copyable` na balíku `Achilex.runtime`.



Obrázok 6: Diagram používaných rozhraní.

Rozhrania `Copyable` a `Discardable` je možné vygenerovať z Paris-u. Rozhranie `Copyable` je používané pri rušení referencií na buffer v objektoch, ktoré reprezentujú hodnoty terminálov a neterminálov a sú uložené v interných štruktúrach syntaktického analyzátora. Teda pre objekty, ktoré implementujú toto rozhranie, sa pri volaní metódy `zzClearReferences`, zavolá metóda `parserCopy`. Pri generovaní balíku `Achilex.runtime` je možné vygenerovať triedu `ScanString` s implementáciou tohto rozhrania. Samotná metóda `parserCopy` je v takom prípade implementovaná pomocou metódy `copy`.

Rozhranie `Discardable` je používané pri zotavovaní z chýb. Pri tejto akcii môže totiž dôjsť k zahadzovaniu symbolov bez toho, aby boli hodnoty symbolov predané programátorovi v akciách. Napríklad keď je pre hodnoty symbolov použitý pool, nedošlo by pri zahodení symbolu k vráteniu tejto hodnoty do poolu. Pre objekty, ktoré implementujú rozhranie `Discardable`, je pred zahodením symbolu zavolaná metóda `parserDiscard`. Pri generovaní balíku `Achilex.runtime`, je možné vygenerovať triedu `ScanString` s implementáciou tohto rozhrania. Samotná metóda `parserDiscard`, je v takom prípade implementovaná pomocou metódy `freeInstance`.

Vykonávanie týchto akcií je možné zapnúť uvedením príkazu `implements` v špecifikácii syntaktického analyzátora. Základné nastavenie vyzerá takto:

```
implements instanceof discardable Object;
implements instanceof copyable Object;
```

Príkaz `instanceof` hovorí, že pred vykonaním príslušnej operácie, sa má vykonať kontrola, či daný objekt implementuje potrebné rozhranie. Na konci príkazu je uvedený zoznam objektových typov v triede `StackRecord`, na ktoré sú mapované typy pre neterminály a terminály a pre ktoré sa majú dané operácie vykonávať. Namiesto príkazu `instanceof`, je ešte možné použiť príkaz `cast`, pri ktorom sa nevykonáva kontrola, či objekt implementuje rozhranie, ale je priamo použitá konverzia na toto rozhranie. Poslednou možnosťou je nešpecifikovať na tomto mieste žiaden príkaz, pričom v takom prípade nedôjde ani k pretypovaniu na rozhranie, ale je priamo zavolaná operácia na objekt.

8.4.1 Príklad použitia generovaných rozhraní

V tejto kapitole boli zatiaľ uvedené dva príklady. Prvý príklad mal ukázať využitie balíku `Achilex.runtime` na prácu s reťazcami. V ďalšom príklade bola predstavená manipulácia s rozhraním poskytovaným syntaktickým analyzátorom. Tento príklad bol založený na príklade spracovania zoznamu stavov lexikálneho analyzátora uvedenom v 6. kapitole. V tomto záverečnom príklade bude uvedené využitie balíku `Achilex.runtime` s možnosťou vytvárania referencií na buffer v kombinácii s použitím vygenerovaných rozhraní. Tento príklad bude opäť založený na príklade zo 6. kapitoly.

Lexikálny analyzátor

Vytvorený balík `Achilex.runtime` bude podporovať pool bufferov s možnosťou vytvárania referencií. Syntaktický analyzátor bude podporovať rušenie referencií na buffer. Aby bolo možné zabezpečiť tieto požiadavky, je nutné pri generovaní lexikálneho analyzátora použiť nasledujúce parametre:

```
java  -jar Achilex.jar
-genlib
-libdir Achilex/runtime
-libpkg Achilex.runtime
-bufref
-genbufpool
-pref ReferencesCleaner
-genstrpool
-strresizepool
-genrcl
-rcldir Achilex/runtime
-rclpkg Achilex.runtime
-strimplcopy
-copyi Paris.Copyable
-strimpldiscard
-discardi Paris.Discardable
spec.alex
```

Je vidieť, že oproti prvému príkladu počet parametrov značne narástol. K novým dôležitým parametrom patria `bufref`, ktorý povoľuje vytváranie referencií na buffer a `genbufpool`, ktorý zabezpečí vytvorenie poolu bufferov. Opäť je použitý pool inštancií triedy `ScanString`. Parametre obsahujúce v názve `rcl`, sa postarajú o vygenerovanie rozhrania `ReferencesCleaner`. Parameter `pref` hovorí, že syntaktický analyzátor bude implementovať rozhranie `ReferencesCleaner` a je ho možné volať pri uvoľňovaní referencií na buffery. Parametre `strimplcopy` a `strimpldiscard` hovoria, že trieda `ScanString` bude implementovať rozhrania `Copyable` a `Discardable`. V špecifikácii lexikálneho analyzátora je potrebné doimportovať vygenerovaný balík:

```
import Achilex.runtime;
```

a uviesť nasledujúci parameter

```
%usebufpool
```

Tento parameter zabezpečí použitie správneho inicializátora pre buffery s poolom. Je takisto možné uviesť kapacitu poolu, pomocou parametra `%pool` a limit naalokovaných znakov, kedy bude dochádzať k uvoľňovaniu referencií na buffer, parametrom `%poolalloc`. Ďalej je potrebné deklarovať pool inštancií triedy `ScanString`.

```
private ScanStringPool strPool = new ScanStringPool(256);
```

Použitie referencií na buffer je možné demonštrovať na predaní reťazca identifikátora syntaktickému analyzátoru. Lexikálna akcia má nasledujúci tvar:

```
{LETTER} ({LETTER}|{DIGIT})*
{
  yyparser.yysetTokenStatic("IDENTIFIER");
  yyparser.yysetTokenLeftLocation(yyline + 1,
  yycolumn + 1);
  yyparser.yysetTokenRightLocation(yyline + 1,
  yycolumn + yylength());

  yyparser.yysetTokenValueScanString(
  strPool.getInstance(yybuffer, yybegin, yyend)
  );

  return IDENTIFIER;
}
```

Syntaktickému analyzátoru je vrátená inštancia triedy `ScanString` získaná z poolu, ktorej znakové dáta sú zdieľané so vstupným bufferom.

Syntaktický analyzátor

Na vygenerovanie syntaktického analyzátora sú použité nasledujúce parametre:

```
java -jar Paris.jar
-genifaces
-ifacesdir Paris
-ifacespkg Paris
-copyipool Achilex.runtime.ScanBufferPool
spec.par
```

Parameter `genifaces` spôsobí vygenerovanie rozhraní `Copyable` a `Discardable`. V špecifikácii je potrebné doimportovať balík `Achilex.runtime`:

```
import Achilex.runtime;
```

Ďalej je potrebné uviesť nasledujúce parametre:

```
implementsrcl;
usercl ScanBufferPool;
discardable Paris.Discardable;
copyable Paris.Copyable;
```

Prvý parameter spôsobí, že trieda syntaktického analyzátora bude implementovať rozhranie `ReferencesCleaner`. Nastavením druhého parametra dôjde k vygenerovaniu metódy `zzclearReferences`, ktorej parametrom je inštancia triedy `ScanBufferPool`. Pre objasnenie nasleduje výťah z implementácie volania metódy na rušenie referencií. V tele metódy `getInstance` poolu bufferov, je umiestnený nasledujúci kontrolný blok:

```
if (countFree == 0 && allocSize >= maxAllocSize &&
buf.getInfo().getParser() != null) {
  buf.getInfo().getParser().zzclearReferences(this);
...
```

V prípade, že v poole nie sú žiadne buffery a bol prekročený limit naalokovaných znakov pre buffery, zavolá sa metóda `zzclearReferences`, ktorú poskytuje syntaktický analyzátor. Metóda `zzclearReferences` je v syntaktickom analyzátore implementovaná takto:

```
public void zzclearReferences(ScanBufferPool yypool) {
  while (yypool.isEmpty() && zzcleanTos <= zztos) {
    zznormStacksHolder.copyValue(zzcleanTos, yypool);
    ++zzcleanTos;
  }
...
```

Teda dôjde k postupnému volaniu metódy `copyValue` na symboly na zásobníku, až kým nedôjde k uvoľneniu nejakého buffera do poolu. Zostáva ešte špecifikovať aké symboly budú umožňovať uvoľňovanie referencií. To je možné určiť príkazom `implements`

`copyable`, ktorý už bol popísaný. Deklarácia symbolov by pre náš príklad so spracovaním zoznamu stavov vyzerala takto:

```
terminal                    LEFT, AST, RIGHT;
terminal     ScanString     IDENTIFIER;
non terminal StateList      state_list, ident_list;
static       String name;
locations    int line, int column;

implements cast discardable Object;
implements cast copyable Object;
```

Keďže všetky objektové typy pre terminály a neterminaly implementujú rozhrania `Copyable` a `Discardable`, nie je potrebné v príkazoch `implements` uvádzať kľúčové slovo `instanceof`, ale stačí použiť `cast`. Samotné pravidlá na vytváranie kolekcie `StateList` by mali rovnaký tvar ako v druhom príklade, preto nie sú uvádzané. Ešte je dôležité uviesť, akým spôsobom bude implementovať trieda `StateList` rozhrania `Copyable` a `Discardable`. Implementácia triedy by mohla vyzerať takto:

```
public class StateList implements Copyable, Discardable {
  public List<ScanString> elements;

  public Copyable parserCopy(ScanBufferPool yypool) {
//prejdi zoznam a na každý prvok zavolaj metódu parserCopy
//až kým nie je uvoľnený nejaký buffer
  }

  public void parserDiscard() {
//prejdi zoznam a na každý prvok zavolaj metódu parserDiscard
  }
}
```

Je takisto nutné podotknúť, že po skončení práce s každou inštanciou triedy `StateList`, je nutné explicitne zavolať metódu `parserDiscard`, aby došlo k uvoľneniu referencií na buffer a vráteniu inštancií triedy `ScanString` späť do poolu.

9 Porovnanie generátorov parserov

Porovnanie prebehlo formou testovacích programov. Porovnávane boli analyzované generátory parserov s vlastnou implementáciou. Testy prebehli na dátach v jazykoch HTML a Java. Na testovanie boli použité tieto verzie generátorov parserov: JavaCC 4.0, JFlex 1.4.1, pre CUP 11a beta 20060330, Achilex 1.1.3 a Paris 1.3.1. Pre vygenerované parsery boli použité nasledujúce parametre:

- podpora vstupných dát v Unicode.
- automatické počítanie pozície nájdených reťazcov.
- výstupný zdrojový kód kompatibilný s Javou 1.5. CUP túto možnosť nepodporuje.
- pre Paris a CUP, bolo vypnuté predávanie informácií o pozíciách neterminálom.
- pre Paris a CUP, bol počet tokenov, ktorý sa musí úspešne spracovať aby mohlo byť zotavenie z chyby úspešné, nastavený na 1. Bolo tak umožnené rovnaké chovanie ako v JavaCC.
- základná veľkosť buffera lexikálneho analyzátora, bola nastavená na 8 192 znakov.

Kvôli získaniu rovnakých výsledkov, boli zdrojové kódy vygenerovaných parserov upravené nasledujúcim spôsobom. V JavaCC bolo pri výpočte pozícii nájdených reťazcov odstránené spracovanie znaku tabulátora ako sekvencie bielych znakov. Pre JFlex a Achilex bolo zrušené spracovávanie špeciálnych Unicode znakov, ktoré reprezentujú konce riadkov, a ktoré v JavaCC nie sú uvažované.

9.1 Testy na HTML dátach

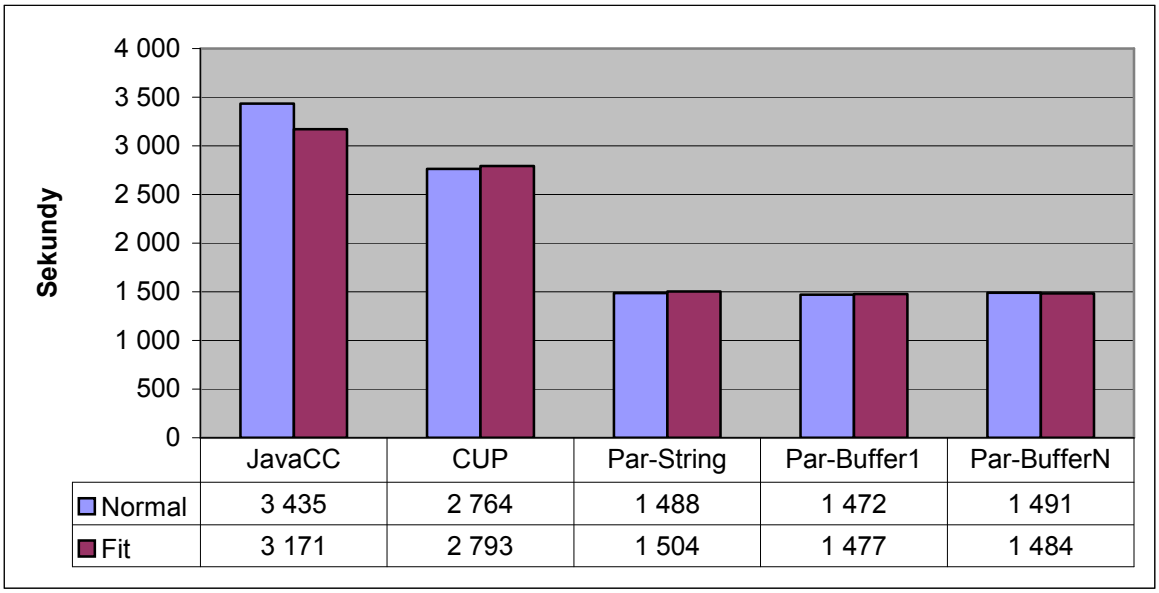
Použitá bola špecifikácia HTML jazyka pre JavaCC, ktorá je k dispozícii na adrese [7]. Gramatika neanalyzuje obsah elementov, ale iba ich štruktúru. Použitie exaktnej gramatiky, ktorá by odpovedala DTD popisu, by spôsobovalo problémy pri testovaní na reálnych HTML dátach. Tie totiž často tejto štruktúre neodpovedajú. K najčastejším problémom patria prehodené alebo úplne chýbajúce zatváracie elementy. Podľa tejto špecifikácie boli vytvorené aj špecifikácie pre JFlex/CUP a Achilex/Paris. Gramatika umožňuje zotavenie z chýb, ktoré vzniknú v štruktúre elementu. Typicky sa jedná o nekorektné znaky v názvoch elementov, prípadne atribútov. Zotavenie z chýb zaručuje, aby pri ich vzniku nedošlo k prerušeniu analýzy celého súboru, ale len aktuálneho elementu. Keďže k chybám dochádzalo v testoch často, boli chybové hlášky potlačené. Testy prebehli na dvoch sadách dát: správne štrukturovaných a reálnych.

9.1.1 Správne štrukturované HTML dáta

Testy prebehli na 400 súboroch s HTML dátami o celkovej veľkosti 8,1GB. Veľkosť súborov sa pohybovala od 10MB do 28,4MB. Cieľom testov bol zápis textového obsahu elementov (`PCDATA`) do výstupného súboru. Po získaní reťazca textu od lexikálneho analyzátora, bol text v akcii syntaktického analyzátora zapísaný do výstupného súboru. Po spracovaní súboru došlo k reštartu lexikálneho a syntaktického analyzátora a zrušeniu obsahu výstupného súboru. Test potom pokračoval v spracovávaní ďalšieho súboru.

Počas behu testov dochádzalo k expanzii buffera lexikálneho analyzátora. Keďže JavaCC používa iný algoritmus expanzie buffera ako JFlex a Achilex, a pri inicializácii dochádza k jeho vráteniu na pôvodnú veľkosť, boli vytvorené dve verzie testov. Verzia *Normal*, so základnou veľkosťou buffera 8 192 znakov a verzia *Fit* s veľkosťou 131 072 znakov, pričom v druhej verzii nedochádzalo k expanziám buffera.

V testoch pre JavaCC dochádzalo k vytváraniu inštancií triedy `String` pre všetky generované tokeny. V testoch pre JFlex/CUP dochádzalo k vytváraniu reťazcov iba pre zapisované textové dáta. Pre Achilex/Paris boli vytvorené tri varianty. V testoch *Par-String*, boli textové dáta reprezentované pomocou inštancií triedy `String`. V testoch *Par-Buffer1* bola použitá varianta s jedným bufferom s možnosťou vytvárania referencií. Pre reťazec textu bola použitá reprezentácia pomocou triedy `ScanString`, pričom inštancie triedy sa udržiavali v poole. Tieto testy si vystačili s 2 inštanciami triedy `ScanString`. V testoch *Par-BufferN* bol na rozdiel od testov *Par-Buffer1*, použitý pool bufferov s kapacitou 1MB znakov. Táto kapacita postačovala, takže nedochádzalo k rušeniu referencií volaním metódy `zzclearReferences`, ani k zahadzovaniu bufferov.

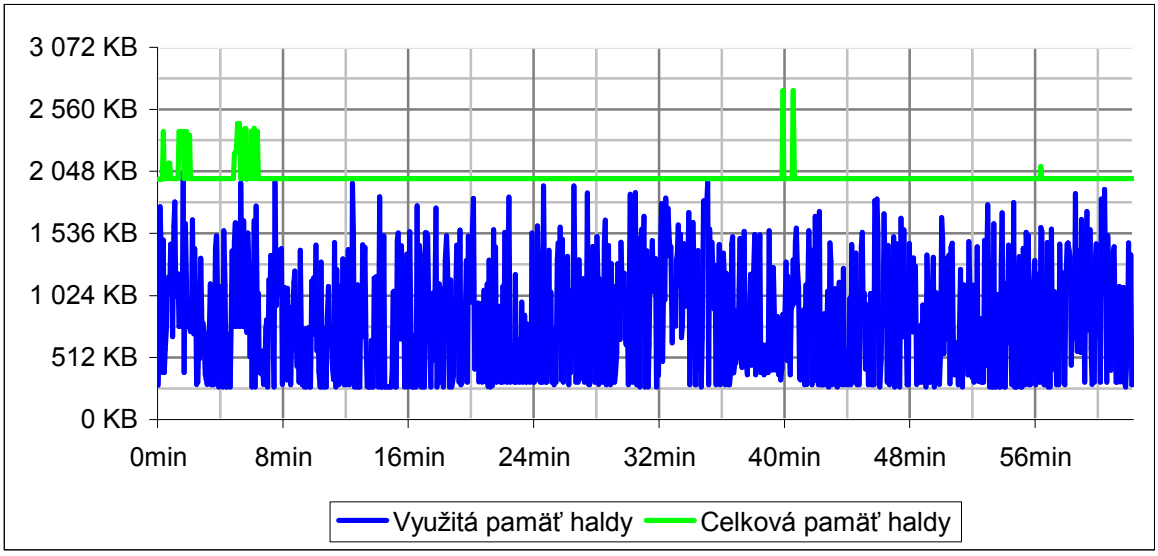


	JavaCC	CUP	Par-String	Par-Buffer1	Par-BufferN
Normal	3 435	2 764	1 488	1 472	1 491
Fit	3 171	2 793	1 504	1 477	1 484

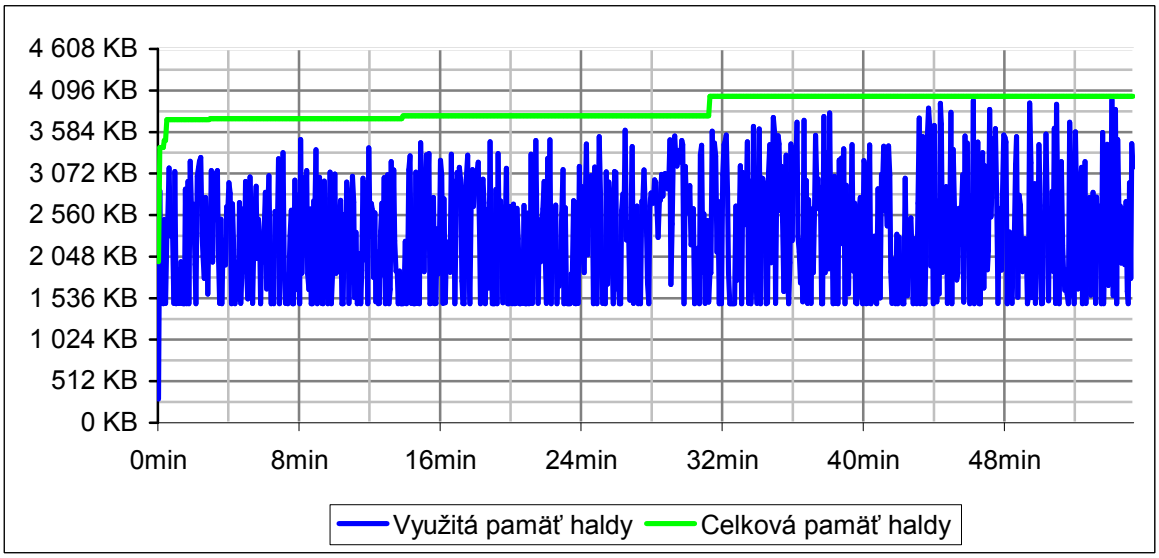
Graf 8: Dosiahnuté časy pre jednotlivé testy.

Testy pre JavaCC dopadli z pohľadu doby behu najhoršie. Testy pre Achilex/Paris sú vo variante *Normal* asi 2,3x a vo variante *Fit* asi 2,1x rýchlejšie. Takisto je zaujímavé porovnanie doby behov medzi variantami *Normal* a *Fit*. Kým v testoch pre JFlex/CUP a Achilex/Paris sú rozdiely minimálne, v prípade JavaCC došlo vo variante *Fit* k zrýchleniu o 8,3%.

Na nasledujúcich grafoch je zobrazené využitie pamäte pre testy *JavaCC/Normal* a *JavaCC/Fit*. V teste *JavaCC/Fit* je vidieť znateľne vyššie využitie pamäte. Okrem väčšej veľkosti buffera, tu hrá úlohu aj použitie polí na uchovávanie pozícií znakov v buffery. V testoch dochádzalo k častému výskytu major collections garbage collectora. Dochádza k častému zaplneniu haldy a následnému veľkému uvoľneniu pamäte. V obidvoch prípadoch došlo k zvýšeniu celkovej veľkosti haldy.

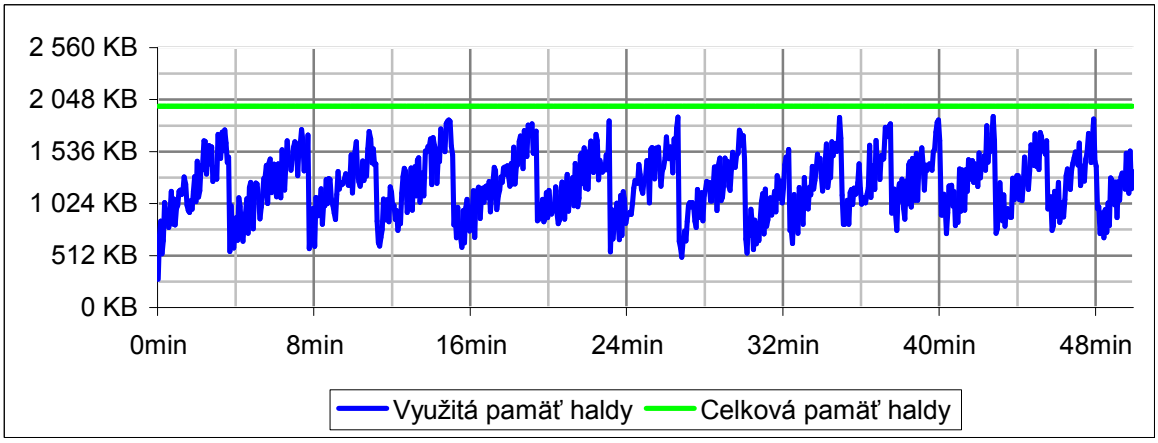


Graf 9: Využitie pamäte pre test JavaCC/Normal.



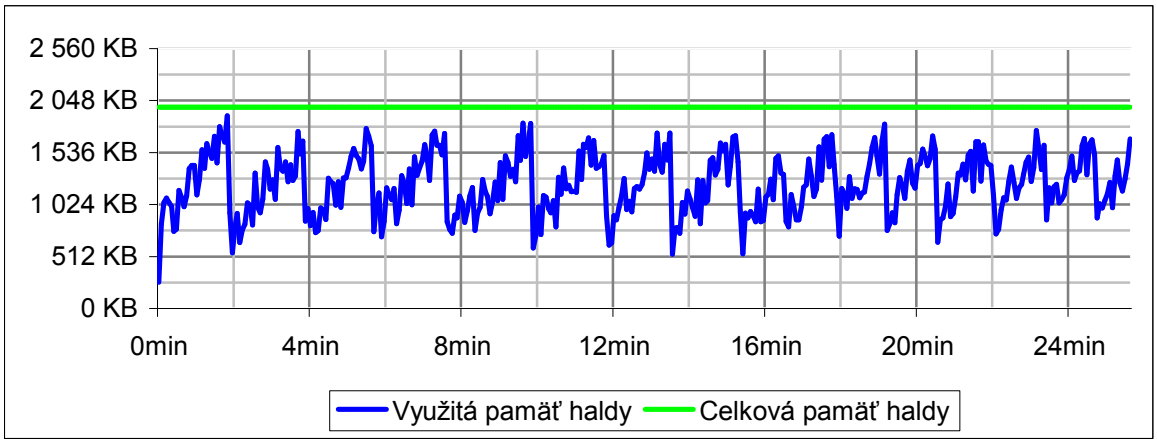
Graf 10: Využitie pamäte pre test JavaCC/Fit.

Testy pre Achilex/Paris dopadli najlepšie. Oproti verziám pre JFlex/CUP boli asi 1,9x rýchlejšie. Na nasledujúcom grafe je znázornené využitie pamäte pre test *CUP/Normal*. V teste dochádzalo k častým minor collections a občasným major collections.



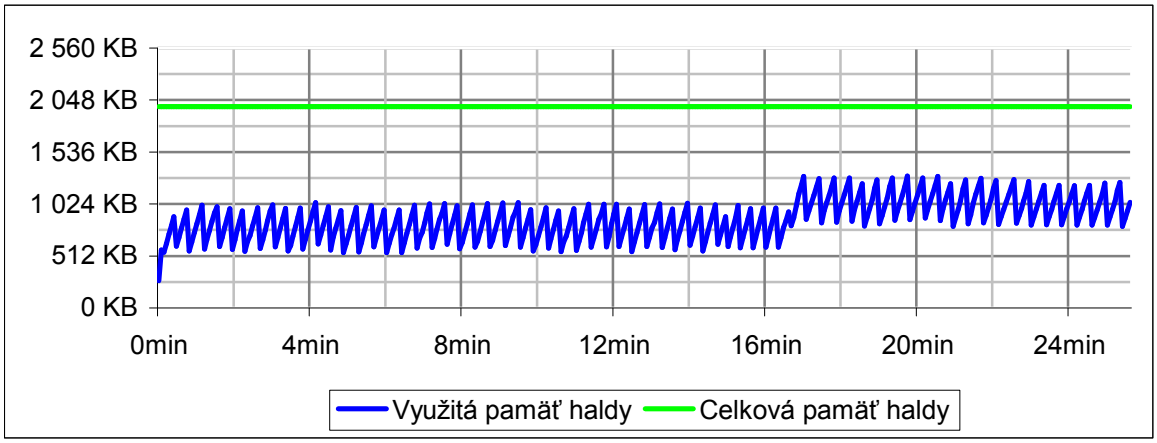
Graf 11: Využitie pamäte pre test CUP/Normal.

Aj napriek tomu, že v testoch *Par-Buffer1* a *Par-BufferN* nedochádzalo k vytváraniu inštancií triedy `String` pre zapisované textové dáta, sú doby behov pre tieto testy skoro identické s testom *Par-String*. Na nasledujúcom grafe je znázornené využitie pamäte pre test *Par-String/Normal*. Graf má podobný priebeh ako v prípade testu *CUP/Normal*.

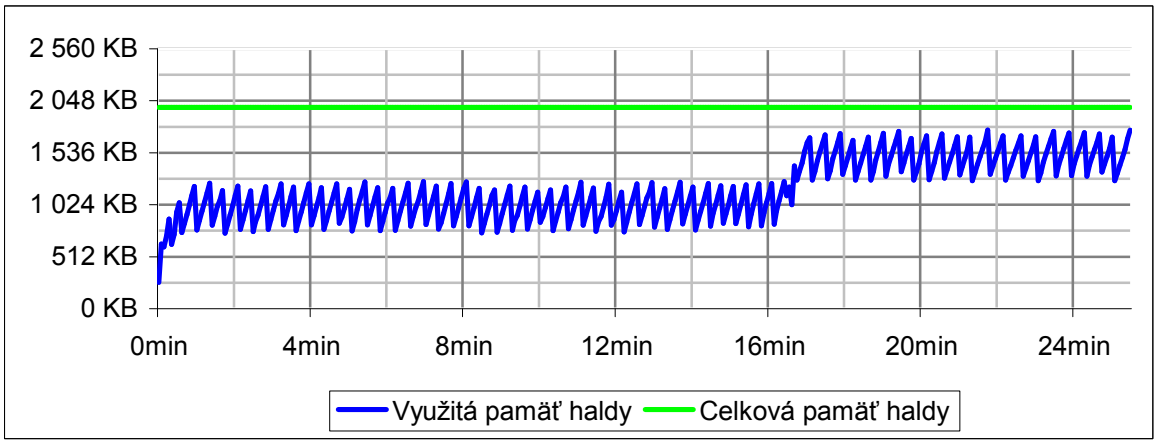


Graf 12: Využitie pamäte pre test Par-String/Normal.

Na nasledujúcich grafoch je znázornené využitie pamäte pre testy *Par-Buffer1/Normal* a *Par-BufferN/Normal*. Oproti testu *Par-String/Normal* je vidieť, že využitie pamäte je stabilné a nedochádza k major collections. Minor collections boli spôsobené vytváraním objektov v štandardných vstupno/výstupných knižniciach Javy. Nárasty vo využití pamäte okolo 16. minúty, boli spôsobené expanziami buffera. Zvýšená veľkosť využitej pamäte pre test *Par-BufferN/Normal* je spôsobená prítomnosťou viacerých bufferov. Konkrétne pre tento test boli potrebné 2 buffery.



Graf 13: Využitie pamäte pre test Par-Buffer1/Normal.



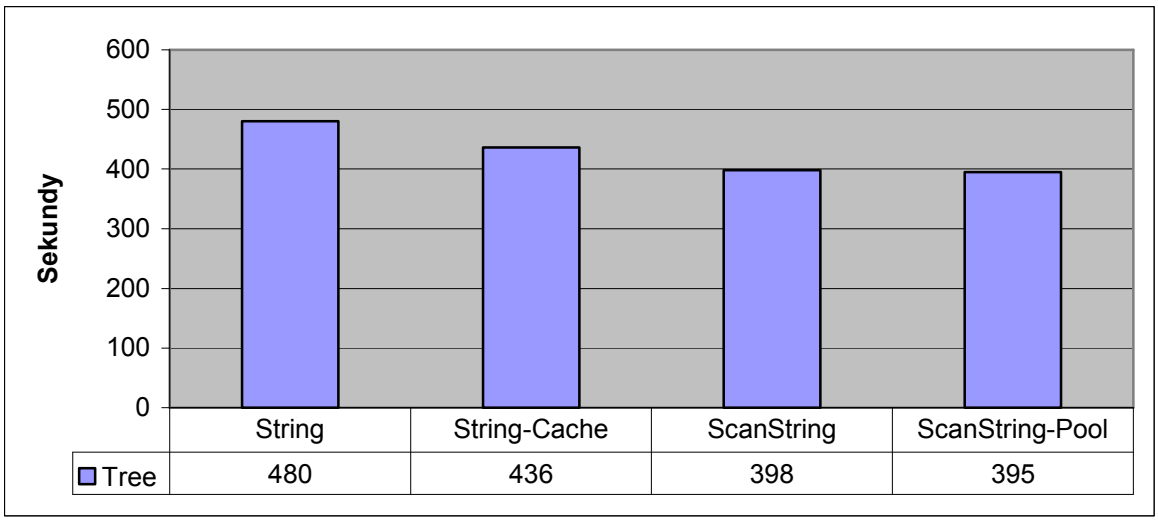
Graf 14: Využitie pamäte pre test Par-BufferN/Normal.

9.1.2 Stromová reprezentácia HTML dát

Úlohou týchto testov bolo ukázať využitie poolu bufferov, ktorý je implementovaný v Achilex-e. Predchádzajúce testy ukázali, že hoci použitím poolu bufferov došlo k zníženiu aktivity garbage collectora, výsledné časy boli skoro identické ako vo variantách bez poolu. Testy prebehli znova na správne štruktúrovaných HTML dátach. Cieľom testov bolo vytvorenie „plytkého“ stromu pre každý HTML dokument. Pod koreňovým uzlom reprezentujúcim dokument, boli napojené jednotlivé elementy. Pod každým otváracím elementom bol napojený zoznam atribútov. Ku generovaniu stromu dochádzalo v syntaktickom analyzátore. Pre každý uzol stromu bol vytváraný nový objekt. Po vytvorení stromu, došlo k jeho prejdeniu a zápisu hodnôt atribútov `src` pre elementy `img` do výstupného súboru. Po spracovaní HTML dokumentu bol strom zahodený. V lexikálnom analyzátore dochádzalo ku generovaniu reťazcov pre všetky nekonštantné

tokeny (medzi konštanty patrí napríklad úvodný znak elementu <), ktoré sa stali súčasťou uzlov stromu. Aby bolo zabezpečené vytváranie dostatočne veľkých stromov, boli zo súborov s HTML dátami vyextrahované HTML dokumenty s veľkosťou aspoň 100 000 bytov. Tieto HTML dokumenty boli organizované v súboroch o veľkosti 25MB. Testy prebehli na 60 súboroch o celkovej veľkosti 1,5GB.

Testy prebehli iba pre Achilex/Paris. Porovnávané boli implementácie bez využitia poola bufferov, kde dochádzalo v lexikálnom analyzátore k vytváraniu inštancií triedy `String` pre rozpoznané reťazce a implementácie s využitím poola bufferov, kde na reprezentáciu reťazcov bola použitá trieda `ScanString`. Oproti štandardnému vytváraniu inštancií triedy `String`, bola testovaná aj verzia s využitím keše, kde veľkosť keše bola nastavená na 8 192 znakov. Pre verzie s poolom bufferov boli vytvorené dva testy. Test *ScanString*, v ktorom boli vytvárané vždy nové inštancie triedy `ScanString`, pričom znakové dáta reťazca boli zdieľané s bufferom a test *ScanString-Pool*, kde boli inštancie udržiavané v poole. Veľkosť buffera bola nastavená na 131 072 znakov. Kapacita poolu bola nastavená na 1MB a nedochádzalo k rušeniu referencií volaním metódy `zzclearReferences`, ani k zahadzovaniu bufferov. Testy si vystačili so 4 buffermi v poole. V teste *ScanString-Pool* bolo poolovaných 30 033 inštancií triedy `ScanString`.



Graf 15: Dosiahnuté časy pre jednotlivé testy.

Podľa výsledkov je test *ScanString* oproti testu *String* rýchlejší približne o 21%. Napriek tomu, že v teste *ScanString-Pool* bol použitý pool inštancií triedy `ScanString`, je čas skoro identický ako v prípade testu *ScanString*. Využitie keše prinieslo v teste *String-Cache* oproti testu *String* zrýchlenie o 10,1%.

9.1.3 Reálne HTML dáta

Testy prebehli na 100 súboroch s HTML dátami o celkovej veľkosti 2,3GB. Veľkosť súborov sa pohybovala od 19,6MB do 24,8MB. Cieľom testov bol zápis textového obsahu elementov spolu s ich váhou do výstupného súboru. Váha textu bola priradená na základe najbližšieho uvažovaného elementu obaľujúceho daný text. Uvažované boli nasledujúce elementy:

- **`TITLE`** – textovému obsahu bola priradená váha *n*.
- **`H1 – H6`** – textovému obsahu bola priradená váha *n/x*, kde *x* je číslo od 1 do 6.
- **`A`** – textovému obsahu bola priradená váha 1.

Číslo *n* bolo určené vstupným parametrom testu.

Popis implementácie

Keďže reálne dáta neboli správne štruktúrované, bolo nutné zabezpečiť správne rozpoznanie konca obalu, aj v prípade chýbajúceho alebo nesprávneho zatváracieho elementu. Za týmto účelom, bol v syntaktickom analyzátore implementovaný zásobník elementov.

Po rozpoznaní otváracieho elementu je zistené, či môže byť obsahom elementu na vrchole zásobníka. Na zisťovanie prípustného obsahu elementov, sú pre všetky známe elementy vytvorené statické hašovacie tabuľky, s množinou prípustných elementov. V prípade, že sa jedná o neprázdny element a podmienka je splnená, dôjde k jeho vloženiu na vrchol zásobníka. Pri nesplnení tejto podmienky, sú od vrcholu zásobníka prechádzané elementy, až kým sa nenarazí na element, pre ktorý je rozpoznaný element prípustný. Potom sú prejdené elementy zo zásobníka odstránené. V prípade, že sa na taký element nenarazí, je nájdený element zahodený.

```
<a name="check">
<font class=sblkhdr></font>
<font class="mbody">
The software on your computer is often ...
<p>
```

Pre tento príklad bude otvárací element `p` pochopený ako ukončenie elementu `a`, a teda text za týmto elementom už nebude braný do obsahu elementu `a`.

Po rozpoznaní zatváracieho elementu, je tento element porovnaný s otváracím elementom na vrchole zásobníka. V prípade, že sa elementy zhodujú, je otvárací element zo zásobníka odstránený. V opačnom prípade sú prechádzané elementy od vrcholu zásobníka, až kým sa nenarazí na otvárací element zhodujúci sa s nájdeným zatváracím elementom. Potom sú prejdené elementy zo zásobníka odstránené. V prípade, že sa na taký otvárací element nenarazí, je nájdený zatvárací element zahodený.

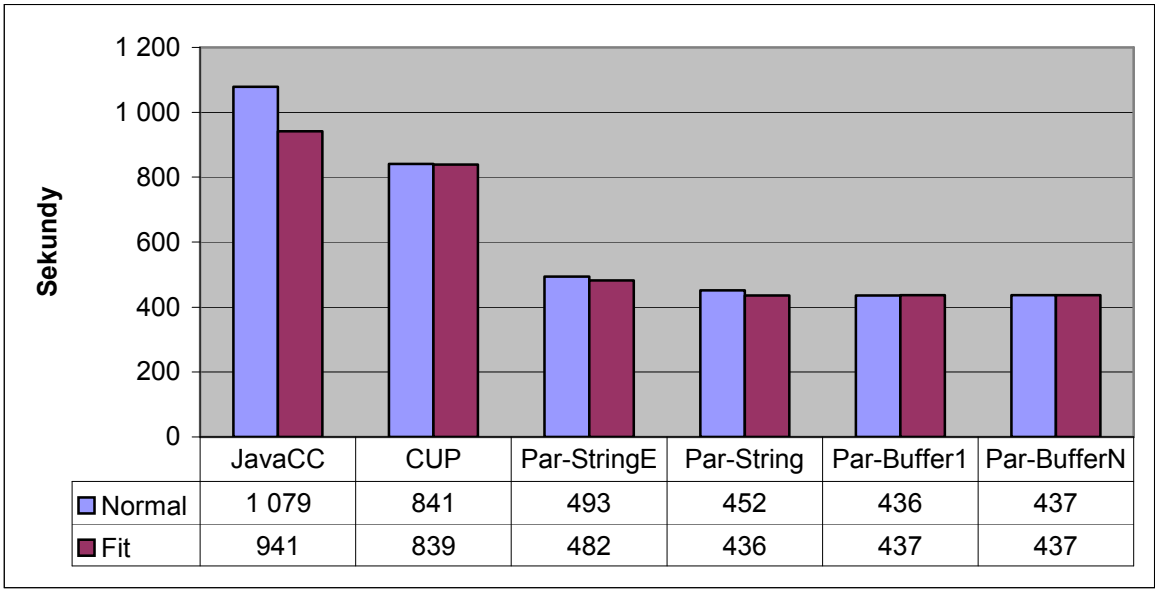
```
<font face=arial size=-2> ... <a href=r/1l>Answers</font>
```

V tomto príklade uzatvorí zatvárací element `font` aj element `a`, a teda ďalší text už nebude braný ako obsah elementu `a`.

Okrem generovania reťazcov pre textový obsah, dochádzalo v lexikálnom analyzátore ku generovaniu číselných identifikátorov rozpoznaných elementov. Pre tento účel bola vytvorená hašovacia tabuľka známych elementov. Neznáme elementy sa do tabuľky postupne pridávali. Po skončení spracovania súboru, bol obsah tabuľky nahradený pôvodnou známou množinou elementov. Syntaktický analyzátor teda nepracoval s reťazcami elementov ale s ich číselnými identifikátormi.

Z pohľadu veľkosti bufferov, využitia poolov a generovania reťazcov pre textový obsah, zostala implementácia testov rovnaká ako na správne štruktúrovaných dátach. V testoch pre JavaCC a JFlex/CUP dochádzalo v lexikálnom analyzátore k vytváraniu inštancií triedy `String` pre rozpoznané elementy. Rozpoznaný reťazec bol potom skonvertovaný na veľké písmena a z hašovacej tabuľky bol získaný jeho číselný identifikátor. V testoch pre JavaCC, bolo nutné na reprezentáciu elementov vytvoriť novú odvodenú triedu tokena s novým atribútom, ktorý uchovával číselný identifikátor. V testoch pre Achilex/Paris nedochádzalo k vytváraniu inštancií triedy `String` pre elementy. Konverzia rozpoznaného reťazca na veľké písmená, bola implementovaná volaním metódy z API knižnice, ktorá zabezpečila priamu konverziu reťazca zo vstupného buffera do znakového poľa, ktoré bolo uchovávané počas celej doby behu testu. Nad znakovým poľom bola potom vytvorená inštancia triedy `ScanString`, pričom inštancie boli udržiavane v poole. Následne bol získaný identifikátor reťazca z hašovacej tabuľky. Výnimkou boli testy *Par-StringE*, kde bola použitá implementácia s generovaním inštancií triedy `String` jednak pre elementy, ale aj pre textový obsah.

Vyhodnotenie výsledkov



	JavaCC	CUP	Par-StringE	Par-String	Par-Buffer1	Par-BufferN
Normal	1 079	841	493	452	436	437
Fit	941	839	482	436	437	437

Graf 16: Dosiahnuté časy pre jednotlivé testy.

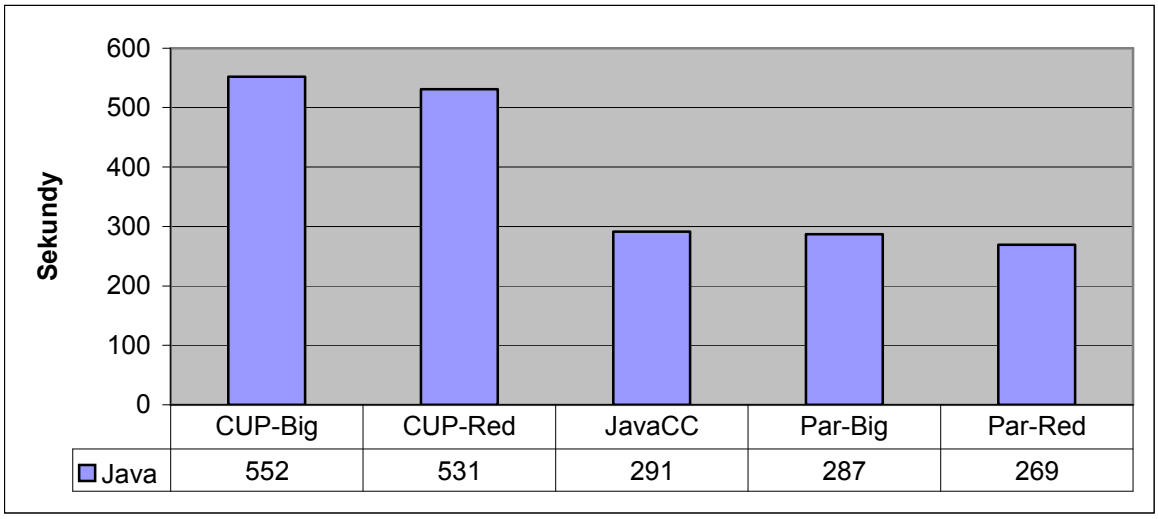
Pri porovnaní časov je vidieť, že výsledky sú podobné ako v testoch pre správne štrukturované HTML súbory. Testy pre Achilex/Paris *Par-Buffer1* a *Par-BufferN*, sú vo variante *Normal* asi 2,5x a vo variante *Fit* asi 2,2x rýchlejšie ako testy pre JavaCC. Vo variante *Fit* testov pre JavaCC došlo k zrýchleniu o 14,7%. Oproti testom pre JFlex/CUP, sú testy *Par-Buffer1* a *Par-BufferN* asi 1,9x rýchlejšie. Oproti testom *Par-StringE* sú testy *Par-Buffer1* a *Par-BufferN* vo variante *Normal* rýchlejšie o 13% a vo variante *Fit* o 10,5%.

9.2 Testy na zdrojových súboroch Javy

Porovnanie prebehlo na špecifikácii pre verziu 1.5. Pre JavaCC bola špecifikácia získaná priamo z repozitára JavaCC na adrese [8]. Pre JFlex a Achilex bola špecifikácia lexikálneho analyzátora vytvorená transformáciou zo špecifikácie pre JavaCC. Pre CUP bola použitá gramatika, ktorá je k dispozícii na adrese [9]. Táto gramatika bola použitá aj pre Paris.

Na testovanie bolo použitých 10 852 súborov so zdrojovými kódmi Javy o celkovej veľkosti približne 106,5MB. Zdrojové kódy boli spojené do väčších súborov o veľkosti 20MB. Na určenie ukončenia zdrojového kódu bola použitá špeciálna znaková sekvencia, ktorá sa inde v dátach nenachádzala. Aby bolo možné vytvoriť väčší objem dát, vo výsledných súboroch sa zdrojové kódy v rôznych permutáciách opakovali. Vytvorených bolo 50 súborov o celkovej veľkosti 1GB.

Do použitých gramatík bolo pridané zotavenie z chýb, takže pri vzniku chyby nedôjde k prerušeniu analýzy celého súboru, ale iba spracovávaného zdrojového kódu. Cieľom testov bola analýza týchto súborov. Na výstupe sa zobrazovali chybové hlášky informujúce o pozícii vzniknutej chyby. Keďže sa jedná o zložitejšiu gramatiku ako v prípade HTML, bola v testoch pre CUP a Paris použitá aj varianta, ktorá umožňuje zmenšiť veľkosť tabuliek syntaktického analyzátora. V CUP-e sá táto možnosť zapína parametrom `compact_red`, v Paris-e parametrom `compactred`.



Graf 17: Dosiahnuté časy pre jednotllivé testy.

Najhoršie dopadli testy pre JFlex/CUP. Test *Par-Big* je 1,9x rýchlejší ako test *CUP-Big*. Je takisto vidieť zrýchlenie pri použití menších tabuliek v CUP-e a Paris-e. Test *CUP-Red* je približne o 4% rýchlejší ako test *CUP-Big*, test *Par-Red* je oproti testu *Par-Big* rýchlejší o 6,6%. Oproti testom pre HTML, došlo v teste pre JavaCC v porovnaní s ostatnými testami k výraznému zrýchleniu. Rýchlosť testu je porovnateľná s testami pre Achilex/Paris. Test *Par-Red* je oproti testu *JavaCC* o 8,2% rýchlejší.

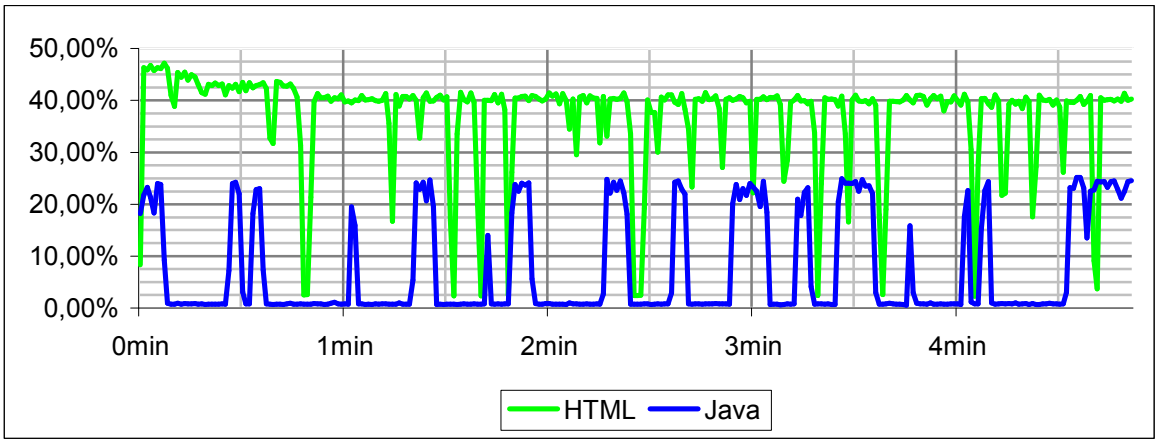
9.3 Analýza testov pre JavaCC

Na základe výrazne odlišných výsledkov pre JavaCC na HTML dátach a zdrojových súborov Javy, bola porovnaná aktivita garbage collectora na testoch *JavaCC/Fit* pre správne štruktúrované HTML dáta a *JavaCC* na zdrojových súboroch Javy. Test *JavaCC/Fit* bol upravený tak, že nedochádzalo k zápisu textového obsahu elementov do výstupného súboru. Na získanie informácií o činnosti garbage collectora boli testy pustené s parametrami `verbose:gc` a `XX:+PrintGCTimeStamps`. Aktivita garbage collectora bola meraná ako percento času stráveného v garbage collectory z 1 sekundy behu testu. Na teste pre HTML dáta je vidieť podľa grafu 18 výrazne vyššiu aktivitu garbage collectora. V HTML gramatike je použité nasledujúce pravidlo na spracovanie elementu:

```
void Element() :
{
}
{
  LOOKAHEAD(2) Tag()
  | EndTag()
  | CommentTag()
  | DeclTag()
  | LOOKAHEAD(2) ScriptBlock()
  | LOOKAHEAD(2) StyleBlock()
  | LOOKAHEAD(2) <TAG_START> <LEX_ERROR>
  | PCDATA
  | EOL
}
```

Zväčšenie výhľadu pred neterminálmi `Tag()`, `ScriptBlock()` a `StyleBlock()` je nutné, pretože všetky pravidlá s týmito neterminálmi na ľavej strane začínajú tokenom `TAG_START`, ktorý reprezentuje úvodný znak otváracieho elementu <. Rozdelenie na viacero pravidiel je nutné, pretože elementy `SCRIPT` a `STYLE` vyžadujú samostatné spracovanie. Ako už bolo uvedené, v JavaCC je použitý spojový zoznam tokenov kvôli uchovávaniu výhľadu. Po preskúmaní zdrojových súborov vygenerovaného parsera, boli v triede parsera objavené pomocné atribúty, `jj_scanpos` a `jj_lastpos` odkazujúce na tento spojový zoznam. K ich použitiu dochádza vo chvíli, keď je použitá konštrukcia `LOOKAHEAD`. Napríklad prvé použitie konštrukcie `LOOKAHEAD` je nahradené volaním nasledujúcej metódy:

```
final private boolean jj_2_1(int xla) {
  jj_la = xla; jj_lastpos = jj_scanpos = token;
  try { return !jj_3_1(); }
  catch(LookaheadSuccess ls) { return true; }
  finally { jj_save(0, xla); }
}
```

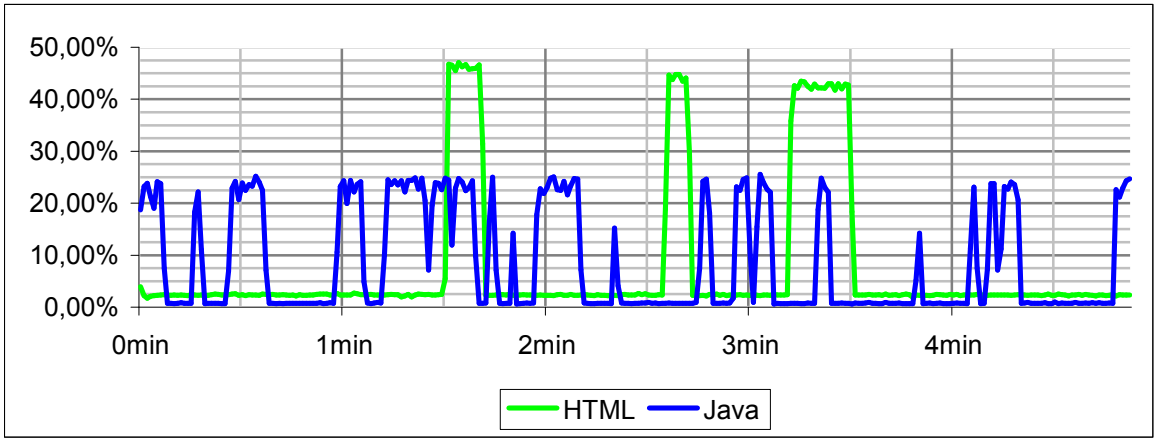


Graf 18: Aktivita garbage collectora pre testy na JavaCC.

Na začiatok tela metódy bol vložený kód, ktorého úlohou bolo zistiť dĺžku spojového zoznamu odkazovaného z atribútov `jj_scanpos` a `jj_lastpos`. Bolo zistené, že dĺžka spojového zoznamu sa niekedy pohybuje v rádoch desiatok až stoviek tokenov.

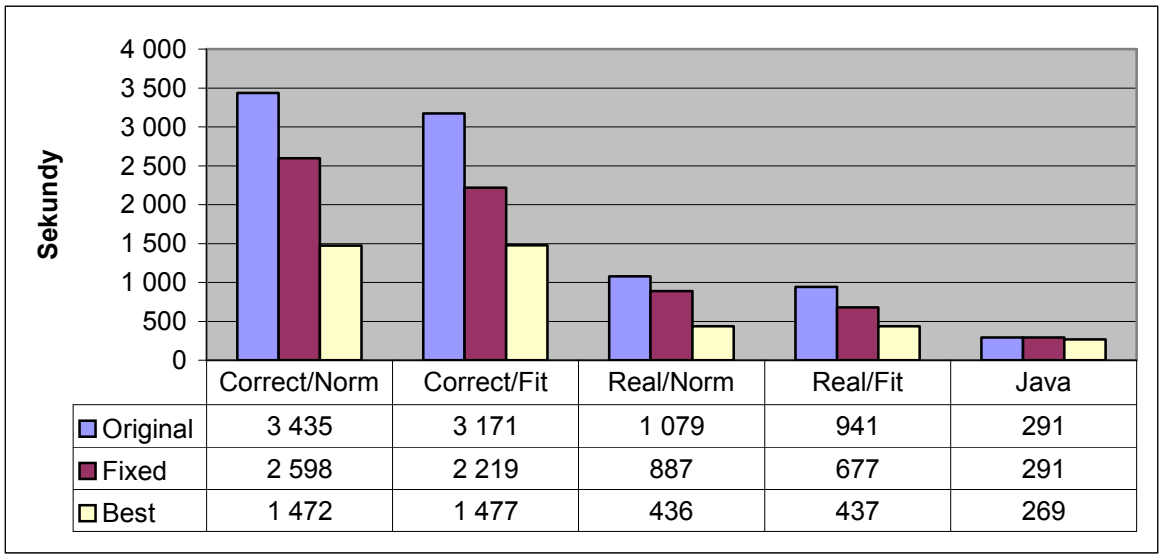
Pravdepodobne teda nedochádza k uvoľňovaniu odkazov na spojový zoznam po skončení použitia týchto atribútov, ale až keď sú použité znova. Medzi opätovnými použitiami týchto atribútov môže uplynúť určitá doba, počas ktorej sa predlžuje životnosť objektov v spojovom zozname a pripájaním ďalších tokenov sa zväčšuje živá časť spojového zoznamu. Tieto faktory môžu spôsobiť, že pri aktivite garbage collectora, dôjde k presunu objektov do old generation a halda sa tak postupne plní, čo vyvoláva časté major collections garbage collectora. V testoch na zdrojových kódoch Javy tieto problémy pozorované neboli.

Pre všetky testy na JavaCC došlo k úprave výsledných zdrojových kódov tak, že bolo doplnené explicitné nastavenie spomínaných atribútov na hodnotu `null` v miestach, kde pravdepodobne došlo k ukončeniu ich použitia. Správnosť týchto úprav nie je zaručená, ale na použitých testoch boli pozorované rovnaké výstupy ako v pôvodných verziách. Najprv je pre porovnanie uvedená aktivita garbage collectora na testoch z predchádzajúcej časti.



Graf 19: Aktivita garbage collectora pre testy na JavaCC po úpravách.

Na nasledujúcom grafe sú znázornené časy pre jednotlivé testy na JavaCC z predchádzajúcich častí.



	Correct/Norm	Correct/Fit	Real/Norm	Real/Fit	Java
Original	3 435	3 171	1 079	941	291
Fixed	2 598	2 219	887	677	291
Best	1 472	1 477	436	437	269

Graf 20: Dosiahnuté časy pre jednotllivé testy.

Testy *Original* označujú pôvodné verzie pre JavaCC. V testoch *Fixed* došlo k uvedenej úprave zdrojových kódov pre JavaCC. Testy *Correct/Norm* a *Correct/Fit* odpovedajú testom na správne štrukturovaných HTML dátach. Testy *Real/Norm* a *Real/Fit* odpovedajú testom na reálnych HTML dátach. Testy *Best* označujú najrýchlejšie testy pre Achilex/Paris z danej kategórie.

Je vidieť, že po úprave zdrojových kódov došlo na HTML dátach k výraznému zrýchleniu. Všetky testy predbehli prípadne sa priblížili svojim ekvivalentom pre JFlex/CUP. Testy *Fit* pre Achilex/Paris sú oproti verziám *Fixed* približne o 50% rýchlejšie.

Ďalšie rozdiely v časoch medzi JavaCC a JFlex/CUP, Achilex/Paris vzhľadom k testovanému jazyku, je možné hľadať v povahe samotného jazyka a takisto v použitej gramatike. Povaha jazyka ovplyvňuje množstvo a veľkosť generovaných reťazcov v lexikálnom analyzátore pre JavaCC. V Jave je oproti HTML veľká časť jazyka pokrytá konštantnými reťazcami (kľúčové slová). Takisto veľkosť reťazcov je malá, keďže sa väčšinou jedná o názvy premenných, metód, atď.

JFlex/CUP a Achilex/Paris používajú na implementáciu lexikálneho a syntaktického analyzátora konečné automaty reprezentované ako tabuľky čísel. Oproti tomu JavaCC používa konštrukcie programovacieho jazyka ako volanie metód, switch bloky atď. Použitie veľkých tabuliek môže negatívne vplývať na efektívnosť pri prístupe k pamäti. Jedná sa o spomínaný princíp lokality. Je vidieť, že pri zmenšení tabuliek syntaktického analyzátora použitím parametra `compactred`, došlo k určitému zrýchleniu v testoch na zdrojových súboroch Javy.

10 Záver

Cieľom tejto práce bolo vytvoriť efektívny LALR(1) generátor parserov pre jazyk Java, ekvivalentný s existujúcimi produktmi Flex a Bison. Keďže v prostredí Javy už takéto nástroje existujú (JFlex a CUP), ukázalo sa rozumné založiť implementáciu nového generátora parserov na týchto nástrojoch. JFlex a CUP naviac disponujú vlastnosťami, ktoré sa v pôvodných nástrojoch Flex a Bison nenachádzajú. Či už sa jedná o podporu Unicode alebo automatický výpočet pozícií tokenov vo vstupnom súbore.

Cieľom implementácie bola úprava spomínaných programov, za účelom zníženia pamäťových a časových nárokov vygenerovaného parsera. Na základe výsledkov testov je možné konštatovať, že tento cieľ splnený bol. Ako ukazujú testy, využitie techník implementovaných vo vytvorenom generátore parserov je závislé na konkrétnej úlohe, ktorú je potrebné riešiť.

Časť implementovaných techník, ako je napríklad object pooling, vyžadujú dodatočnú spoluprácu zo strany programátora, čo do určitej miery redukuje komfort samotného programovania.

Významnosť a efektívnosť vytvoreného produktu je limitovaná a ovplyvnená vývojom v oblasti implementácie objektových technológii ako je garbage collector a vývoja hardwaru.

11 Literatúra

[1] *Tuning Garbage Collection with the 5.0 Java[tm] Virtual Machine*
http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html

[2] Dennis M. Sosnoski: *Java performance programming*
http://www.javaworld.com/javaworld/jw-11-1999/jw-11-performance-p3.html

[3] Brian Goetz: *Java theory and practice: Garbage collection and performance*
http://www-128.ibm.com/developerworks/library/j-jtp01274.html

[4] *JavaCC - Java Compiler Compiler*
https://javacc.dev.java.net/

[5] *JFlex - The Fast Scanner Generator for Java*
http://jflex.de/

[6] *CUP - LALR Parser Generator in Java*
http://www2.cs.tum.edu/projects/cup/

[7] *JavaCC HTML Parser*
http://www.quiotix.com/downloads/html-parser/

[8] *Repository of JavaCC grammars*
https://javacc.dev.java.net/servlets/ProjectDocumentList?folderID=110

[9] *Java 1.5 CUP Grammar*
http://people.csail.mit.edu/jhbrown/javagrammar/index.html

[10] *The comp.compilers newsgroup*
http://compilers.iecc.com/

12 Prílohy

12.1 Obsah CD

Na CD disku, priloženému k textu práce, sa nachádzajú nasledujúce adresáre:

- **`/src`** - v tomto adresári sa nachádzajú distribúcie Achilex-u a Paris-u. Súčasťou distribúcií sú zdrojové kódy, dokumentácie a výsledné Java archívy `Achilex.jar` a `Paris.jar`. Priložené sú aj pôvodné, nezmenené dokumentácie k JFlex-u a CUP-u.
- **`/test`** - zdrojové kódy testovacích programov.
- **`/text`** - text diplomovej práce.

12.2 Inštalácia a spustenie programov

Inštalácia spočíva v skopírovaní obsahu adresára `src` z CD. Na spustenie programov je potrebné mať nainštalované prostredie J2SE Runtime Enviroment (JRE) vo verzii 5.0. Programy je možné spustiť nasledujúcimi príkazmi:

```
java –jar Achilex.jar [options] file
java –jar Paris.jar [options] file
```

Java archív `Achilex.jar` sa nachádza v adresári `/src/achilex-1.1.3/lib`. Java archív `Paris.jar` sa nachádza v adresári `/src/paris-1.3.1/dist`. V sekcii `options`, je možné uviesť rôzne parametre určujúce podobu výsledného lexikálneho a syntaktického analyzátora. Popis parametrov je možné nájsť v priložených dokukumentáciách. V parametri `file`, je uvedený názov súboru obsahujúceho špecifikáciu.